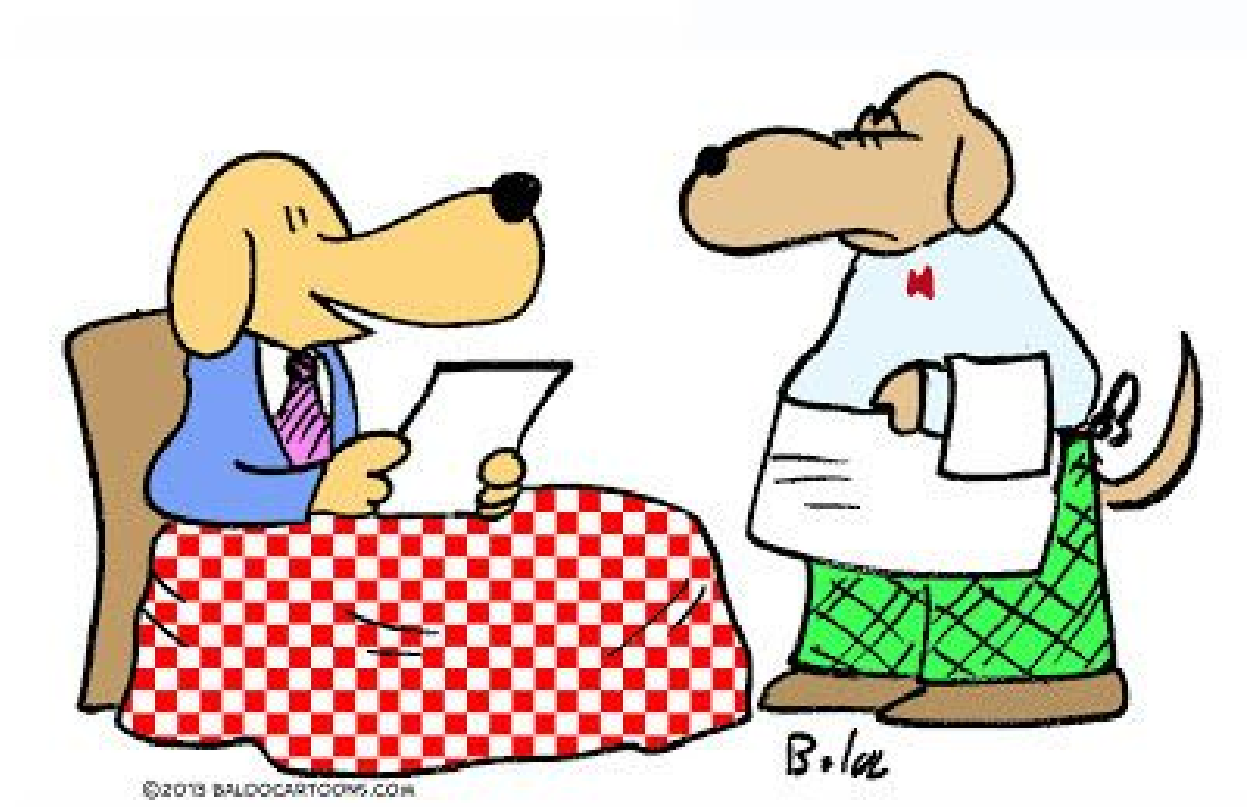
SWE: Serve with Ease

GROUP #1
Project: Restauranting Made Easy
Report 3



Team Members: Annie Antony, Athira Haridas, Christina Parry, Emma Roussos, Christina Segerholm, and Nishtha Sharma

332: 452 Software Engineering, Professor Ivan Marsic

Github URL: https://github.com/powerpuffprogrammers

Division of Work

Athira:

- Requirements Specification :
  - 100% for the following:
    - On Screen Requirements, System Architecture, System Design,, Hardware Requirement, Architectural Styles, Identifying Subsystems, Mapping Subsystems to Hardware, Persistent Data Storage, Network Protocol, Global Control Flow, Execution Orderliness, Time Dependency, Concurrency, User Effort Estimation - Changes we made for each of these
  - 50% of the following:
    - Summary of Changes, OCL Contracts, Data Structures
    - Use Case 1: Casual Description and Fully Dressed Description, System Operation Contracts, Interaction, Diagram, Design Patterns and Principles (under Interaction Diagrams), Changes we made
- Coding :
  - 50 % of the following:
    - HostTableScreen.java, LogInScreen.java,WaiterOneTicketScreen.java, WaiterTickListScreen.java

  100% of the following: IntegrationTestHost.java , ManagerScreen.java,
- Debugging :
  - 100% of the following:
    - HostTableScreen.java, LogInScreen.java,WaiterOneTicketScreen.java, WaiterTickListScreen.java, ManagerScreen.java
- Report Preparation :
  - Formatted Report 3 layout- 33%, delegated tasks amongst group-50%
- Other :
  - Overall revisions, readMeIntegrationTest - 80%

Annie:

- Requirements Specification :
  - 100% for the following:
    - Non-Functional Requirements, Data Types & Operation Signatures
    - Use Case 2: System Sequence Diagram, System Operation Contracts, Design Principles, Changes we made
  - 50% of the following:
    - Design of Test Cases, Summary of Changes
- Coding :
  - ChefOneTickScreen.java - 90% , ChefPanel.java - 50%, IntegrationTestWaiter.java-100%
- Debugging :
  - ChefInterface.java - 50%, ChefOneTickScreen.java - 90% , ChefPanel.java - 50%
- Report Preparation :
  - 100% of the following: Summary of Changes
- Other :
  - readMeIntegrationTest - 15%

Christina S:

- Requirements Specification :
  - 100% of the following:
    - Glossary of Terms, Mathematical Model, Domain Analysis: Domain Model Diagram, Concept Definitions, Association Definitions, Attribute Definitions, Algorithms - Changes we made for each of these
  - 50% of the following:
    - Use Case 1: Casual Description and Fully Dressed Description, System Operation Contracts, Interaction, Diagram, Design Patterns and Principles (under Interaction Diagrams), Changes we made for each of these
    - OCL Contracts
  - 90% of the following: Future Work
- Software Design :
  - 100% of the following:
    - Designed overall layout of system and architecture, creator of distributed databases and message controller, Major Ideas behind System Design and Database distribution, Created GUI template that all interfaces would use.
- Coding :

    - 100% of the following packages:
        - Configuration, DatabaseA, DatabaseB, DatabaseC, MessageController,
    - 100% of the following classes:
        - ChefInterface.java, ChefMessageListener.java, ChefMessageSEnder.java, HostInterface.java, HostMessageListener.java, HostMessageSender.java, TabletApp.java, ManagerInterface.java, ManagerMessageListener.java, ManagerMessageSender.java, DatabaseLoadFRomFileTesting.java,MessageTesting.java, TicketToMessageConversion.java, DatabaseCListener.java, DatabaseCSender.java, DishForTicket.java, KeypadScreen.java, WaiterInterface.java, WaiterMessageListener.java, WaiterMessageSender.java
    - 50 % of the following classes:
        - HostTableScreen.java, LogInScreen.java,WaiterOneTicketScreen.java, WaiterTickListScreen.java
- Debugging :
    - 100% of the following:
        - All backend work (message controller, databases, message sending, ticket sending, interface communication)
- Report Preparation :
    - Formatted Report 3 layout- 33%, delegated tasks amongst group-50%
- Other :
    - 100% of the following:
        - readme.txt, Technical Document, readmeUnit.txt, readMeData.txt, Organized and Maintained github repository
    - Overall Revisions, readMeIntegrationTest-5%, skits-50%

Emma:

- Requirements Specification :
    - 100% for the following:
        - Use Case 3 : Casual Description, System Operation Contracts, Interaction Diagram, Design Patterns and Principles (under Interaction Diagrams), System Sequence Diagram
    - 50% for the following:
        - Design of Test Cases, Use Case 3: Fully Dressed Description, Changes we made
- Report Preparation:
    - 100% for the following:
        - Inserted all information having to do with Use Case 3

Christina P:

- Requirements Specification :
    - 100% of the following:
        - Traceability Matrices and Descriptions, Project Management and History of Work, Demo 2 Brochure, Revised CSR
    - 50% of the following:
        - Use Case 3 : Changes to Use Case 3, Fully Dressed Description, Design of Test Cases
    - Added to Summary of Changes
- Report Preparation :
    - Revisions on CSR and Use Case 3, Formatting

Nishtha:

- Requirements Specification :
    - 100% of the following :
        - Design Patterns , Key Accomplishments , Functional Requirements , Class Diagrams and Interface Specification (Class Diagram, Key, Descriptions) , User Interface Design and Implementation, Use Case 2: Casual Description and Fully Dressed Description, Interaction Diagram, Design Patterns (under Interaction Diagrams)
    - Data Structures - 50%
    - Future Work - 10%
- Coding :
    - ChefPanel.java - 50%, ChefOneTickScreen.java - 10%
- Debugging :
    - ChefInterface.java - 50% , ChefOneTickScreen.java - 10% , ChefPanel.java - 50%
- Report Preparation :
    - 100% of the following :

        - Labeled and captioned all Figures in report , Presentation for Demo 2 , UML Diagrams document , User documentation , Formatted OCL Contracts into tables, Formatted On-Screen Requirements
    - 50 % of the following :
        - ReadMe.txt for Demo 2 , Skits
    - Formatted Report 3 overall layout - 33%
- Other : Overall Revisions

Summary of Changes

Changes in Project Objectives:

- Saving tickets in text files for future reference/ auditing
- Broadcast messaging system from manager to all employees
- Enabling couponing and gift cards

Changes in Functional Requirements*:*

- Head Chef, Host, Manager, and Waiter have been edited with more functional requirements
- Head Chef: editing status of the dishes, notifying the manager, deleting the ticket
- Host: selecting an active waiter to seat
- Manager: deleting a message
- Waiter: notifying other waiters, modifying dishes, editing orders, gift cards or coupons

Changes in On Screen Requirements*:*

- For general employee, we added the part that an employee should not be able to log on to two different tablets at one time.
- For head chef, rather than changing the color for a completed ticket, we have it disappear from the screen.
- For host, we changed the interface to show two lists of tables one for empty and one for occupied/dirty. We also made it required that the host selects a waiter along with the table when seating a party.
- For waiter, similar to chef, the ticket now disappears of the screen when completed. We have yet to implement the owner interface, it is a plan for the future iterations.

Changes in Non Functional Requirements

- Deleted the last non-functional requirement

Changes in Use Case Descriptions:

- UC-1 Closing a Table
    - Changed “on check” status to “paid” to ensure that table is ready to be cleaned
    - Host screen includes a list of waiters to select from and the number of tables each waiter currently has seated
    - Host Interface holds all seating information such as active tables rather than Database B
    - Database B only holds complete list of tables, no status
- UC-2 Finishing a Ticket
    - Replaced previous use case of “Low Inventory” to “Finishing a Ticket”
    - Use case is now from waiter sending the ticket to the chef to waiter picking up hot food
- UC-3 Placing an Order
    - Replaced the “hot food” notification with a normal notification.

Changes in Domain Model Analysis

- We did not include a domain model diagram in our last report so the whole diagram is new as well as the key.

Changes in Concept Definitions

- Reorganized the chart to put all of the containers first then the doers.
- Removed all of the employee even listeners as all events that happen shall be handled by the employee interfaces.

- Took out the low inventory checker since he was no longer needed. Now the inventory updater will perform the threshold check when it updates an ingredients inventory.
- Took out concepts that dealt exclusively with sending or receiving tickets. A ticket can be sent as a message and therefore does not need its own special functions or concepts to handle transmitting it.

Changes in Association Definitions

- A ticket is now sent through the Waiter Message Sender as a message.
- During the event of low inventory, the Inventory Updater will be the one who checks the threshold. Then it will generate a message for the Chef Message Sender to send to the Message Controller. Then the Message Controller will be able to forward it to the manager to notify them.

Changes in Attribute Definitions

- Ticket now includes waiter's name and ID.
- Table now includes whether it is a booth or a table.
- Added Ingredient, Dish, Employee, and Menu concept.
- Rewrote each employee's message sender, message listener, and interface description in a more clear and concise way.
- Changed Host Interface to separate table's by their status.
- Changed Chef Interface to separate ticket by their status.
- Merged inventory updater and low inventory checker.

Changes in System Operation Contracts

- Use Case 1
    - Instead of changing the color of the ticket on the waiter screen we actually remove it from the list as they should no longer be able to access it again since it is paid.
- Use Case 2
    - Instead of Low Inventory, we discussed the actions involved in Finishing a Ticket. This changed all sections of the System Operation Contracts.
- Use Case 3
    - Waiter doesn't need to have a button to create a ticket. A new ticket should already exist once they get sat a table so all they have to do is click on that specific table.

Changes in Mathematical Modelling

- Replaced the host event listener with host interface. The host interface will listen for buttons clicked by the host and also handle requests from the host message listener.
- Added special functions to handle a host pressing the cleaned button and a host pressing the seat button.
- Since we are sending Recently Sat messages to waiters, we needed to add the listOfWaiters so the host interface can get the waiter's id from the waiter's name and pass the id to the message controller so it can send it to the right waiter.

Changes in Interaction Diagrams

- Use Case 1
    - Added the actual names of the functions into the interaction diagram.
    - Changed and took out Database B messages about seating a table. This is no longer needed as the Host Interface will hold all of this information. Database B will hold the list of tables but not their active statuses.

    - Changed the on check status to a paid status. On check meant that a table has not paid yet but paid means that they have already paid and they will be ready to leave soon.
    - Added a list of waiters to the host screen which holds not only the names of waiters that can be sat but also the amount of active tables each waiter currently has. We had to make sure to update this when statuses of tables were changed.
    - Implemented the Command Pattern Design principle. The message controller acts as the command in the model.
    - Explained the socket communication more thoroughly.
- Use Case 2
    - Replaced our previous Use Case idea (Low Inventory) with Finishing a Ticket, upon the feedback provided.
    - The Use Case is entirely different than before.
    - The interaction diagram is also changed now to reflect this new case.
    - Descriptions of actions were replaced with functions existing in code.
- Use Case 3
    - Changed void openTicket(int TableNumber) to void openTicketScreens(int tableNumber) between Waiter and Waiter Interface.
    - Changed Display Order to Display Ticket between Waiter Interface and Waiter.
    - Changed void addDish(Ticket t, Dish d) to void addDishToTicket(Dish dish) between Waiter and Waiter Interface.
    - Changed void waiterMessageListener(Message messToWaiter) to void decodeMessage(Message m) between Message Controller and Waiter Interface. Similarly, we also changed void chefMessageListener(Message messToChef) to void decodeMessage(Message m) between Message Controller and Head Chef Interface.
    - Changed decreaseStock to decrementInventoryForDish between Head Chef Interface and Database C
    - Changed void startDish(Ticket t, Dish d) to updateStatus(Dish dish) between Head Chef and Head Chef Interface.Similarly, we also changed void updateDish(Ticket t, Dish d) to updateStatus(Dish dish) between Head Chef and Head Chef Interface.
    - Modified void sendMessage(Message hotFood) to void sendMessage(Message m) between Head Chef Interface and Message Controller

Changes in System Design

- Subsystems are changed: Database A holds the employee list, Database B holds table queue and Database C holds inventory and menu. Database A will be accessed by all employees, B will be accessed by the host and C will be accessed by chef and waiter.
- For saving employee information and ticket information for future use, we used textfiles with Java rather than SQL. We plan to implement SQL in the future because as of now the amount of information we are storing is limited as we are still in the testing stages of the application.
    - UC-1 Closing a Table
        - Removed Database B messages about table status
        - Implemented the Command Pattern with Message Controller being the command
        - Showed socket communication more thoroughly
    - UC-2 Finishing a Ticket

    - Interactions are now between waiter and chef
    - Implemented Command Pattern
    - Added more interactions between chef interface and chef
- UC-3 Placing an Order
    - Implemented the Command Pattern
    - Reworded the interactions to better suit our code

Changes in Class Diagram

- Deleted functions from Host Interface, Waiter Interface, Head Chef Interface, Message Controller, Manager Interface, DBA Controller.
- Added classes DBB Controller, DBC Controller
- Edited the Class Key Identifier number in the diagram accordingly.
- Edited the placement of classes to accommodate new classes to properly show how they are associated.
- Added functions to Host Interface, Waiter Interface, Head Chef Interface, Message Controller, Manager Interface, DBA Controller, DBB Controller, DBC Controller.

Changes in Algorithms

- Replaced Host event listener with host interface and java swing buttons. Now we use smaller functions for each type of table switch.

Package Diagram Changes

- Updated Databases so that Database A holds employee information, B holds table queue and C holds inventory and menu

Changes we made in Data Types & Operation Signatures

- Since we were not able to implement SQL at this time, we used Java classes to hold data in the databases. The classes are labeled as databaseA,B,C controllers.
- Instead of having functions that would listen for messages from the message controller and send messages to the message controller in each employee interface, we had two separate classes for each employee interface (that were the <EmployeePosition>MessageListener and <EmployeePosition>MessageSender) that deals with communication with the message controller .
- Added JFrame to display the main screens for each employee
- For the chef interface, tickets were separated (based on status) into four ticket queues (unstarted, semi-started, started, finished) , instead of one queue. A hashmap that maps the ticket number to the ticket was also added.
- Added function to load menu items from the database for the waiter
- Add a notify manager function to each employee interface
- Added functions to modify the ticket for the waiter interface
- For the host interface, tables were separated (based on status) into 3 lists (ready, paid, seated), instead of two lists. It also has a list of waiters. It has a loadTables function to load the list of tables from the database for the host.
- For the manager interface, add the mass notification function to send a notification to all the employees.
- For all the employee interfaces, add the addNotification function to add the notification to the current screen.

- Used the separate classes to display the panels (the graphical user interfaces) for the employees using JPanel

Table of Contents:

Customer Statement of Requirements

a. Problem Statement

As a restaurant owner, my customers expect various services to be delivered fast, such as seating and ordering. In order to help my restuarant perform better, I would like a system which automates certain services to assist the employees in their roles. It should improve each employees' organization of their tasks, communication with the rest of the staff, and speed of service to the customer.

The following are problems faced by my employees:

- ➢ Host: Deciding which table to bus next
- ➢ Waiter: Taking and placing orders; Managing a newly sat table, an existing table, or unsatisfied tables
- ➢ Head Chef: Managing inventory, Organizing food preparation
- ➢ Manager: Alerting staff

Additional problems are further described below.

Host / Hostess

Problem 1: *Promptly Bussing Tables*

As a host working in a busy restaurant, one of my main responsibilities is to determine which tables need to be bussed and to convey that information to the busboy. Currently, I look around the restaurant to see if customers are getting up to leave or if any tables are empty and need to be cleaned. This actually takes up a lot of my time that could be spent assisting and seating customers. It is also a little embarrassing when customers that are waiting to be sat point out to me that certain tables aren't clean. I am the first person customers interact with in the restaurant and my top priorities are to make sure customers don't have to wait for a long time and to ensure that they have an enjoyable experience. I would like to have a software application that makes my job easier so I can have a greater focus on the customers. I need to have a way to determine which tables need to be bussed and when. Also, I need the application to show me which tables are ready for customers to be seated. By having a list of tables that are ready to be seated, I will be more organized and can work faster since I won't have to look around the restaurant for empty tables or ones that need to be bussed. Through this system, I will be able to improve how well I do my job.

Waiter

Problem 1: *Taking an order*

Everyone knows a waiter's main job is to take the table's order. It doesn't sound complicated, but when the I'm on my third table, it's hard to stay on top of everything. It's easy to forget to ask a guest ordering the steak how he would like it cooked when I have a million other things on my list to do in the next few minutes. In this case, two situations can occur. In

Scenario A, I take the walk of shame back to the table and ask the question that I forgot. This not only wastes my time but also makes me look unprofessional. Alternatively, in Scenario B, I simply assume that the customer wants their steak done medium, based on popular choice. However, this runs the risk of creating an angry customer who wanted a rare steak and ultimately causing more work for me in the long run. Instead, I would like to have a system that places orders while customers are requesting them. The system would need to have all of the menu items on it. So now when a customer orders an entree, I can open up the menu options, click the entree. Using an application to generate a computer ticket will help me take orders faster which will allow me to focus more on my guests and handle more of them at once.

Problem 2: *Being notified when I get sat/when hot food is ready*

One of the most embarrassing mistakes a waiter can make is not greeting a table within a certain amount of time after the party sits down. When the host seats people at my corner table, it can take me a while before I realize that they have been sat there. Now I have a couple of hungry and angry customers at my table because of my negligence to check my section frequently. It would be great if on the device, there was a notification system that would notify me when I get sat. In order to keep track of what tables I have, I would also like the system to tell me the table numbers of my active tables. This way when I go to place an order, I can see a list of all the tables I have at the moment and click which one I want to place an order for. I also would like to be notified when my table's food is ready to be run out. It is very time consuming to keep walking back and forth into the kitchen checking if any of my tables' food is ready. Instead, I should be notified that my food is ready through the application. This system will not only improve my communication with the kitchen and the host, but it will also clearly specify what I should be doing at the moment.

Problem 3: *Notifying a manager about an unhappy customer*

One of the hardest things to deal with as a waiter is an unhappy customer. No matter what the cause, sometimes I do not have the ability to change their mind. In this case, I have to call my manager over and they take care of the rest. For example, when my table gets their steak rare instead of medium. Cluelessly, I go over and check how everything came out but I get an ear full of complaints from the customer. My next move is to notify my manager, but this may take a while depending on what I am doing at the time, making my customer more unsatisfied. I would like to be able to notify my manager through the application without having to look for him. This communication tool will not only save manager and I precious time, but also will lead to happier customers.

Problem 4: *Taking specific orders for dietary and health restrictions*

Cathy Iko, Carrabba's waiter, realizes that one of the struggles of being a waiter is taking specific orders for customers who have dietary and health restrictions. "If waiters don't pay

attention to specific order details, then an incorrect order could lead to major consequences; like if a customer is vegetarian and their meal unknowingly contained traces of meat, then you have an unhappy customer, and we don't want that." If there is a way for waiters to be able to modify orders to fit customers' health and dietary restrictions, then customers will be happy that their health needs are met to their satisfaction.

Problem 5: *Taking special orders:*

We would also like to consider the convenience and happiness of our customers to ensure that they have a positive dining experience and frequent our restaurant again in the future. We want the waiter to be able to ask for specific dish modifications. The waiter should be able to modify menu options such that they are able to accommodate various needs. The waiter should be able to make a given order have a "To Go" option, give a birthday notification and dessert to a customer, and many more.

Problem 6: *Paying for orders with coupons, gift cards, and promotions*

Carrabba's manager, Sally Kobuta, claims that one of the issues she deals with is allowing customers to pay with coupons and gift cards. "If customers can pay with gift cards and get discounts for coupons, then customers will be satisfied and want to come back to our restaurant again and again." This will make it easier for customers to pay for their orders. If our restaurant honors coupons, gift cards, and promotions, customers will want to come back in the future if they have optimal service and discounted costs of their orders.

Problem 7: *Notifying the manager*

In our solution, we want employees to have clear and concise communication among each other, and so it is imperative that waiters are able to be in direct communication with their manager for any given reason. It is especially important for the waiters to keep communication with the manager to ensure customer satisfaction. This way, waiters can inform them of a specific concern, including but not limited to coupons, issues with service, issues with meal, issues with waiter, etc.

Problem 8: *Allowing waiters in need of assistance to communicate with other waiters*

Deryl Moller, a waiter at Carrabba's, claims that one area of focus is to improve communication between waiters. "I wish there was a way for me to communicate between other waiters. It's hard to clearly communicate while some of us are at opposite ends of a restaurant on a busy Friday night." Often times, waiters bussing other tables see that another table needs attention from its respective waiter. Additionally, if another waiter needs help, waiters can communicate to other waiters for assistance. If our restaurant improves communication in a quick and simple way between waiters, we can ultimately improve internal communication within the restaurant.

Manager

Problem 1*:* *Notifying all employees*

A manager is pretty much the backbone of the restaurant. It is my job to keep all employees updated when we run out of a certain food so they can notify their customers. The sooner I update them, the better the chance of us not having any problems with customers that might try to order that specific food item. But it is not easy trying to track down each waiter. I wind up running around the restaurant and sometimes I can't deliver the information soon enough. If that happens then the waiter and I have to reproach the customers, apologize, and try to see if they would like to change their order or not. Most of the time the customer gets disappointed, frustrated, or angry that they cannot have what they originally wanted. We need a way to send each other notifications very quickly and efficiently. The faster I can get the message out, the better. I would like to ease the communication between the staff.

Problem 2: *Keeping sales history*

As a manager, one crucial aspect of my role is keeping a record of all of the sales for the restaurant. If there was a way for me to electronically store ticket information and access it whenever to observe sales patterns and trends, or to give a customer a refund if necessary. Studying sales trends of the restaurant would help me determine which time of day, what day of the week, which entrees and appetizers are most frequently purchased.

Head Chef

Problem 1**:** *Keeping track of inventory*

The main attractions of a restaurant is the food that it serves. As head chef, my job involves supervising the sous chefs and making sure all the orders have been covered. It doesn't sound so chaotic, however, on weekends, the restaurant is packed with hungry customers. It is easy to lose track of what ingredients are running low, especially when I am busy trying to make sure that the current dishes come out correctly and quickly. Low inventory causes some of the customers to not get the dishes they order. It wouldn't be efficient for me to keep running to the inventory periodically to make sure that we have all the items we need for our dishes. I would like the application to keep track of my inventory and alert me when it is low. Having this would allow me to focus on my main job of running the kitchen.

Problem 2**:** *Keeping track of ticket orders*

As a chef, one of the problems that I run into is how to manage the immense amount of orders I get during rush hours. The current system in use is paper tickets on a rotating ticket holder. This is very unorganized and creates chaos in the kitchen when I have to decide which order is next. I would like a way to organize my incoming tickets in chronological order. It is also difficult to keep track of the progress of multiple dishes for one table. The goal is to have all

dishes come out together. I would like the application to have a system which keeps track of which dishes on the ticket are started. This way I can be sure that the dishes come out together. These two features will increase my organization and mitigate my mistakes.

b. Novelties in our Solution

Our system is not like the other restaurant systems. Our aim is not to automate the whole restaurant process or cut anyone out of a job. Two of our members work as waitresses in local restaurants so our group has a grasp on how a restaurant works. Customers don't come out to eat just for the food. Customers come out to eat for the experience, which includes the food and the service. They want to discuss what looks good on the menu and ask their waiter what his favorite dish is. Our system is going to make each employee's life easier, including the owner's. By providing each employee with a tablet, their tasks will be simplified. This will make each employee less stressed and more customer friendly. Previous systems have taken out the intermediate workers, like hosts and waiters, and replaced them with tablets and podiums. Our solution not only strives for efficiency, but also aims to increase organization and communication within the restaurant. In this case we will not just be increasing the speed of service to the customer but also raising customer satisfaction by taking some of the stress off of the staff.

Host

Promptly Bussing Tables

- The way we go about notifying the host that a table is almost ready to be cleaned (on check/leaving) is different than previous methods.
    - Group 1 in 2013 had an on check method where they would update the corresponding person once the server printed the check.
    - In our method, we have the waiter press a "paid" button after he has taken that table's payment. After this the host is signaled that the table will be leaving soon as it is on check.
        - Our way is more efficient because we cut out the variable wait time between the table receiving the check and the table paying for the check.
- The way we go about displaying the host screen is also different than previous projects.
    - Group 1 in 2013 showed all of the tables under one list under the host screen.
    - In our solution, the host sees two different lists at once. One list is the list of ready tables (empty and clean tables) and another list for seated or on check tables.
        - This is more efficient as the host does not need to scroll or switch screens to view which tables are ready and which ones need to be bussed.

Waiter

Taking an order

- The way we go about having the waiter take an order is different from the implementation of previous groups.
    - Group 4 of Spring 2014 had the waiter place an order on a tablet and send it to the chef.
    - Our method does the same implementation as the above group, but we also go beyond that by asking for specific dish modifications. On the waiter screen, there is a "Modify" button, and upon selection, the waiter may choose modifications to the order, including but not limited to making orders for all dietary restrictions, making orders "To Go", and giving a birthday notification and dessert to a customer.

Being notified when I get sat/when hot food is ready

- The way we notify a waiter when he has recently been sat is different than previous projects.
    - Group 3 of Spring 2015 had the host send the waiter a message when they have been sat.
    - Our solution is to still have that message sent but have multiple notifications. First the message will be displayed as a hanging notification on the waiter's screen until the waiter clicks the acknowledge. A second notification will be placed on the table button that says "Recently Sat". This would not disappear until the table was opened.
        - Our solution is better because it ensures that the waiter has seen the message and knows he is sat. It will keep the waiter reminded until he approaches the table and open the ticket.
- The way we notify a waiter when he has hot food waiting is different than previous projects.
    - Group 3 of Spring 2013 has the chef send a message to the waiter to notify the server he has hot food waiting.
    - Our solution is to still have that message sent but have the message displayed on that tables button in the waiter's screen. On that button there will be a "Hot food" message under the table number. The text field will disappear upon clicking on the table, since opening the ticket implies that the waiter is preparing to take out the hot food.
        - Our solution is better because it ensures that the waiter has acknowledged that he has hot food and is preparing to take it out.

Notifying another waiter that his table requires attention

- ➢ Our method for handling this is novel to our group.
    - We allow waiters to communicate with other waiters so that they can ensure their respective tables are notified about any customer concerns. Waiters can enter a table number to determine the waiter who is bussing a given table and then send a notification to that waiter. This can be used to notify waiters that their tables need immediate attention.

Notifying a manager about unhappy customer

- ➢ Our method for handling unsatisfied customers is different than previous groups'.
    - While many groups did discuss customer satisfaction in terms of wait time but not follow up with the rest of the customer's experience. They assumed that with the application the customers would always be happy with what they ordered.
    - In our solution, we have a "notify manager" button which allows the waiter to send a message to the manager. Upon clicking this button, there are more buttons for the waiter to notify the waiter with a specific concern, including but not limited to coupons, issues with service, issues with meal, issues with waiter, etc.
        - Our way is better because it allows for quick and easy communication between the waiter and the manger, reducing the amount of time the unhappy customer has to wait.
        - Our way also optimizes communication with the manager to specifically voice any concerns a waiter may have.

<u>*Manager*</u>

Notifying all employees

- ➢ Our application for the manager was different than previous projects.
    - Group 3 of Spring 2015 gave the manager the ability to communicate with one waiter at a time.
    - Our solution supplies a broader messaging method. The manager will be allowed to broadcast a message to all employees (host, servers, and chef). Our solution will also provide quick interactions between all employees and the manager by having a button on their screens to call the manager.
        - Our solution is better because it allows for easier communication between employees.

<u>*Head Chef*</u>

Keeping track of inventory

- ➢ The way we go about keeping track of inventory is similar to the implementation of previous groups.

- Group 3 of Spring 2013 kept track of the amount of ingredients in the inventory so the chef could determine what needed to be restocked.
- In our method, we have a similar implementation. We decrement the quantity of the ingredients used in the dish when the chef changes the status to started.

Keeping track of ticket orders

- The way we go about keeping track of ticket orders is different from the implementation of previous groups.
  - Group 4 of Spring 2014 have organized their tickets by chronological order.
  - Our solution uses more states to categorize the progress of a ticket. We break down the organization into two factors:
    - First is separation by ticket progress (Finished, started, semi-started, unstarted)
    - Then by chronological order (based on when the chef received the ticket).
  - Our solution is more organized because having multiple states within the first separation allows the chef to more easily keep track of what dishes need to be prepared next.

Glossary of Terms

- **Bussing a table:** Cleaning a table after the customers have left.
- **Comps:** Complimentary dishes. You would say a manger has comped off the meal if they buy the meal for the customer.
- **Database:** Holds all of the information about the restaurant on it.(Employee information, menu information, inventory information, and table information). Will be run on a desktop computer. All tablets/clients will be able to access data in database. Only certain employees will be able to modify data in the database.
- **Got Sat:** Describes the scenario where a party sits down at a table in a waiter's section. Example: "Jack just got sat at table 32" means that a group of customers has just sat down at table 32 and Jack is their waiter.
- **Head Chef:** Chef that handles delegating all of the orders. Keeps track of what food needs to come out next and tells his chefs what they need to be making. Also in charge of inventory.
- **Head Host:** Greets the customers and gives them the wait time. Handles what table each party should be sat at. Tells busboys or other hosts what table to bus next.
- **Hot Food:** An order that is done and ready to be taken out to a table.
- **Manager:** Runs the show, makes sure customers are happy and that everything is running smoothly.
- **On Check:** This is a phrase used to describe a table. It means that the customer has the check and is waiting to pay.
- **Owner:** Owns the restaurant, has the highest stake in the company. Can also act as a manager.
- **Party:** Group of customers who are dining together (can be party of 1,2,..n).
- **Priority Ticket:** Describes a ticket that should be made as soon as possible by the chef. An example of this would be when a waiter drops a dish and must have it remade.
- **Running food/ Running out food:** Taking dishes that are ready to be served from the kitchen to the table. Delivering food to a table.
- **Ticket:** A list of orders for a specific table. This is what the waiter gives to the chef to tell them what they need to be cooked for a certain table.
- **Waiter:** Waiter or waitress, responsible for good customer service and taking the orders of customers. Also should run out food for their tables.

System Requirements

Functional Requirements

Changes we made in Functional Requirements:

- Head Chef, Host, Manager, and Waiter have been edited with more functional requirements
- Head Chef: editing status of the dishes, notifying the manager, deleting the ticket
- Host: selecting an active waiter to seat
- Manager: deleting a message
- Waiter: notifying other waiters, modifying dishes, editing orders, gift cards or coupons

User Stories

Priority is in descending order where the most important user story is at the top.
Size is how much effort goes into implementing that story.

For the ID of these user stories, the format is:

ST - position initial - requirement type (functional: F) - requirement #

General Employee:

ID	User Story	Size
ST-GE-F-1	As an employee, I should be able to pick up any tablet and log in using my employee ID.	4
ST-GE-F-2	As an employee, I should be directed to the correct screen depending on my position after logging in.	2

Head Chef:

ID	User Story	Size
ST-HC-F-1	As a head chef, I should be able to update the status of the ticket and view the progress order of the tickets.	6
ST-HC-F-2	As a head chef, I should be able to see when a new order is placed.	7
ST-HC-F-3	As a head chef, I should be able to notify the waiter when the order is complete.	6
ST-HC-F-4	As a head chef, I should be updated when the number of items in the inventory for an ingredient is low.	3
ST-HC-F-5	As a head chef, I should be able to see the orders I haven't attended yet.	4
ST-HC-F-6	As a head chef, I should be able to edit the status of a dish whenever I want.	5

ST-HC-F-7	As a head chef, I should be able to notify the manager about any issues.	4
ST-HC-F-8	As a head chef, I should be able to delete a ticket once it is finished.	6

Host:

ID	User Story	Size
ST-H-F-1	As a host, I should be able to view all tables.	4
ST-H-F-2	As a host, I should be able to view the status of any table.	5
ST-H-F-3	As a host, I should be able to change the status of any table.	5
ST-H-F-4	As a host, I should be able to notify a manager about issues with customers.	3
ST-H-F-5	As a host, I should be able to select the waiter I want to seat a table for.	5

Manager:

ID	User Story	Size
ST-M-F-1	As a manager, I can send the waiters notifications about food items that are not available.	4
ST-M-F-2	As a manager, I can view all notifications from the host, waiters, and the kitchen.	3
ST-M-F-3	As a manager, I can pick who I want to send the messages too.	4
ST-M-F-4	As a manager, I can delete messages I have already handled.	3

Waiter:

ID	User Story	Size
ST-W-F-1	As a waiter, I should be able to start a new ticket for a newly seated table.	2
ST-W-F-2	As a waiter, I should be able to reopen an existing ticket for a table.	2
ST-W-F-3	As a waiter, I should be able to enter in any menu items under a table.	4

ST-W-F-4	As a waiter, I should be able to modify dishes when placing an order.	7
ST-W-F-5	As a waiter, I should be able to easily send a ticket to the kitchen when I'm done taking it.	3
ST-W-F-6	As a waiter, I should be able to close a table after they have paid.	4
ST-W-F-7	As a waiter, I should be alerted when I get sat.	3
ST-W-F-8	As a waiter, I should be alerted when one of my table's food is ready.	3
ST-W-F-9	As a waiter, I should be able to notify a manager that something is wrong with a table.	3
ST-W-F-10	As a waiter, I should be able to easily go back and edit an order that I have taken but not placed yet.	3
ST-W-F-11	As a waiter, I should be able to notify another waiter when their table needs them.	5
ST-W-F-12	As a waiter, I should be able to handle coupons or customers buying gift cards.	6

Owner:

ID	User Story	Size
ST-O-F-1	As the owner, I should be able to add and remove employees on the database.	3
ST-O-F-2	As the owner, I should be able to modify employee information on the database.	3
ST-O-F-3	As the owner, I should be able to modify menu item data on the database.	5
ST-O-F-4	As the owner, I should be able to add and remove menu items on the database.	5
ST-O-F-5	As the owner, I should have exclusive access to these features.	3

Non Functional Requirements

Changes we made in Non Functional Requirements:

- Deleted last non functional requirement.

ID	Priority	Description
NF-1	2	The GUI for the staff should be easy to use and not require much time to understand.
NF-2	4	The system must allow for quick communication between devices.
NF-3	4	The system should be able to back up information and read from backed up information.
NF-4	5	The system shall not allow unknown devices to connect to it
NF-5	2	The database system should be scalable in regards to amount of employees and menu items it can store.
NF-6	5	The system should have an authorization subsystem that determines a certain client's permission to the database.

On Screen Requirements

Changes we made in On Screen Requirements:

- For general employee, we added the part that an employee should not be able to log on to two different tablets at one time.
- For head chef, rather than changing the color for a completed ticket, we have it disappear from the screen.
- For host, we changed the interface to show two lists of tables one for empty and one for occupied/dirty. We also made it required that the host selects a waiter along with the table when seating a party.
- For waiter, similar to chef, the ticket now disappears of the screen when completed. We have yet to implement the owner interface, it is a plan for the future iterations

<u>General Employee:</u>

ST-GE-O-	As an employee, I should be able to logout from my screen using a	3

1	logout button.	
ST-GE-O-2	As an employee, I should be able to see who I am logged in as on the screen.	3
ST-GE-O-3	As an employee, when typing in my ID, I would like to see a number pad.	2
ST-GE-O-4	As an employee, I should not be able to log in to two different tablets at once.	3

Chef:

ST-HC-O-1	As head chef, I should be able to update the dish within a ticket by pressing a button.	4
ST-HC-O-2	As head chef, I should be able to view the list of ticket orders on my screen.	2
ST-HC-O-3	As head chef, I should be able to view the contents of the ticket and the status of each dish on my screen.	3
ST-HC-O-4	As head chef, I should be able to change the status of a meal within a ticket with by clicking on that dish's status.	3
ST-HC-O-5	As a head chef, when I view my ticket screen, tickets should be ordered based on their completeness going from all dishes started to none started.	3
ST-HC-O-6	As a head chef, when I complete an order and tap that ticket, it should disappear off my screen	2

Host:

ST-H-O-1	As a host, I should see a list of the tables with the ones available to be sat.	5
ST-H-O-2	As a host, I should be able to update the status of a table from ready to seated by clicking on that table's button.	2
ST-H-O-3	As a host, I should be able to view the maximum occupancy of any table.	2
ST-H-O-4	As a host, the tables should be shown in two lists, one containing	5

	ready tables and one containing paid/seated tables.	
ST-H-O-5	As a host, when selecting a table to seat, I should be prompted to select a waiter to seat that table with.	5

Manager:

ST-M-O-1	As a manager, I would like to be able to create a new message with the push of a button.	2
ST-M-O-2	As a manager, I should be altered on screen whenever I receive a new message.	2
ST-M-O-3	As a manager, after I have viewed a message, it should be deleted from my screen.	2

Waiter:

ST-W-O-1	As a waiter, I would like to be able to click on a button with a table number on it and have it open up the ticket for the corresponding table.	3
ST-W-O-2	As a waiter, I should be able to see all the active tables I have on my home screen.	3
ST-W-O-3	As a waiter, I would like to see when I have been recently sat and when hot food is ready for a table.	4
ST-W-O-4	As a waiter, when selecting dishes to add, I should be able to add and remove dishes to a ticket.	5
ST-W-O-5	As a waiter, when I press paid for a ticket, it should disappear off my screen.	4

Owner:

ST-O-O-1	As the owner, when I log on I should be able to pick whether I want to change the menu, the inventory, or the employee data.	2

ST-O-O-2	As the owner, when I decide what section I wish to change (menu, employee, or inventory) I should have buttons to add, remove, or modify a current item or employee.	4
ST-O-O-3	As the owner, when I am looking at a list of employees or items, I should have a scroll bar to access all of them.	3

Use Cases

Use Case 1 - Closing a Table

Changes we made in Use Case 1:

Changed the on check status to a paid status. On check meant that a table has not paid yet but paid means that they have already paid. It is better to notify the host about a paid table as this party is more likely to leave sooner. We also added a list of waiters to the host screen which holds not only the names of waiters that can be sat but also the amount of active tables each server currently has. We had to make sure to update this when statuses of tables were changed. In user effort estimation we changed the amount of clicks for Host. Instead of having an update status for a table button we just had seat and cleaned buttons. The seat requires one more click to update status because we now need to know the waiter who we are giving a table to. We also took out Database B messages about seating a table. This is no longer needed as the Host Interface will hold all of this information. Database B will hold the list of tables but not their active statuses.

Casual Description

This use case describes the processes that the different staff members must go through when a table is done eating and paid their bill. It starts when the waiter takes the payment of the table and continues all the way through the host re sitting that table with a new party.

Fully Dressed Description

→ What user does

← what system does in response

Related User Stories	ST-H-F-1, ST-H-F-2,ST-H-F-3, ST-H-F-5,ST-H-O-1,ST-H-O-2, ST-H-O-5, ST-W-F-2,ST-W-F-6, ST-W-F-7, ST-W-F-10, ST-W-F-12, ST-W-O-1,ST-W-O-2,ST-W-O-5
Initiating Actor	Waiter

Actor's Goal	Print out the check to give to the table, and take the payment for that check so the party can leave. Notify the host that table has paid.
Participating Actors	Host, Busboy
Pre-Conditions	Check is viewable on-screen and there is a print check button on the bottom of the screen that will print the check at the printer.
Post-Conditions	The table should no longer be shown on the waiter's screen. The Host's screen should be updated to include a paid status for the departing table.
Flow of Events for Main Success Scenario	1→ Waiter opens up the table's ticket by clicking the button on the screen with the corresponding table number on it. 1.1←The system shows the current ticket on screen as well as a print check button on the bottom left. 2→ Waiter clicks print check button on the screen. 2.1←The system sends the check to the restaurants printer to be printed. → Waiter picks up the check from the printer and delivers it to the table. 3→ Waiter takes the payment for the table. He clicks the paid button under that table as that table is done. Then clicks the yes button to the "Are you sure" question. 3.1←The system updates the waiter's software so that the button for that table number is removed from the screen.. 3.2←The system sends a message to the host application notifying the application that this table is on check and leaving soon. This is an indirect message sent first to the message controller who then forwards it to the host. 3.2.1←The system updates the host's screen by changing the status of the table to on check by changing the color of the table to yellow. → Host views his screen with the list of tables and realizes that there is a table that has paid. He notifies a busboy to prepare to bus that table in the near future. → The party leaves and the Busboy cleans the table. 4→ Host acknowledges that table is cleaned and changes the status of that table from paid to ready which changes the color of the table to green.

	4.1←The system shows the updated list of table statuses on the host's screen with the newly cleaned table on list on the right side of the screen. 5→ Host sits the same table again with a different party by using the seat button on his screen along with the waiter's name. 5.1←The system shows the table as seated on the host's screen and that table is moved to the left side of the screen and changes the color of the table to red. It also increments that waiter's number of tables by one. 5.2←The system sends a message to the corresponding waiter's application to notify them they have gotten sat at that table and shows the new ticket on his screen with a "recently sat" message on the bottom of the ticket. This is an indirect message sent first to the message controller who then forwards it to the waiter. 5.3→ Waiter sees that he has gotten sat at that table number on his screen. It is shown on his list of tables screen. He goes to greet it and taps the new table. 5.4←The system removes the "recently sat" from the ticket once the waiter has tapped on it.
Flow of Events for Extension (Alternate Scenarios)	1→ Waiter opens up the table's ticket by clicking the button on the screen with the corresponding table number on it. 1.1←The system shows the current ticket on screen as well as a print check button on the bottom left. 2→ Waiter clicks print check button on the screen. 2.1←The system sends the check to the restaurants printer to be printed. → Waiter picks up the check from the printer and delivers it to the table. A) Waiter loses the check/ check gets dirty/ reprinting check 3→ Waiter opens up the same table's ticket by clicking the button on the screen with the corresponding table number on it. 3.1←The system shows the current ticket on screen as well as a print check button on the bottom left. 4→ Waiter clicks print check button on his screen. 4.1←The system sends the check to the restaurants printer to be printed. → Waiter picks up the check from the printer and delivers it to the table. Progression continues as Main Success Scenario. B) Waiter hits the print button by accident/needs to add something to ticket

	3→ Waiter opens up the same table's ticket. 3.1←The system shows the current ticket on screen as well as options of menu items to add to the list 4→ Waiter orders whatever he had to order by clicking appropriate button 4.1←The system shows the updates the current order on waiter's screen. 5→ Waiter clicks send order button. 5.1←The system sends the order to the chef's application. This is an indirect message sent first to the message controller who then forwards it to the chef. 6→Later, the waiter prints the new check by pressing print check button. 6.1←The system sends the check to the restaurants printer to be printed. → Waiter picks up the check from the printer and delivers it to the table. Progression continues as Main Success Scenario. C) Host changes the status of table to clean before it's cleaned 3→ Waiter takes the payment for the table. He clicks the paid button under that table as that table is done. 3.1←The system removes this ticket from the waiter's list of tickets. 3.2←The system sends a message to the host application notifying the application that this table is paid and leaving soon. This is an indirect message sent first to the message controller who then forwards it to the host. 3.2.1←The system updates the host's screen by changing the status of the table to on check and changes the color of the table to yellow. → Host views his screen and realizes that there is a table paid. He notifies a busboy to keep prepare to bus that table in the near future. 4→ The party leaves and Host mistakenly changes the status of a paid table to ready. 4.1←The system updates the host's screen by changing the status of the table to ready.

System Sequence Diagram

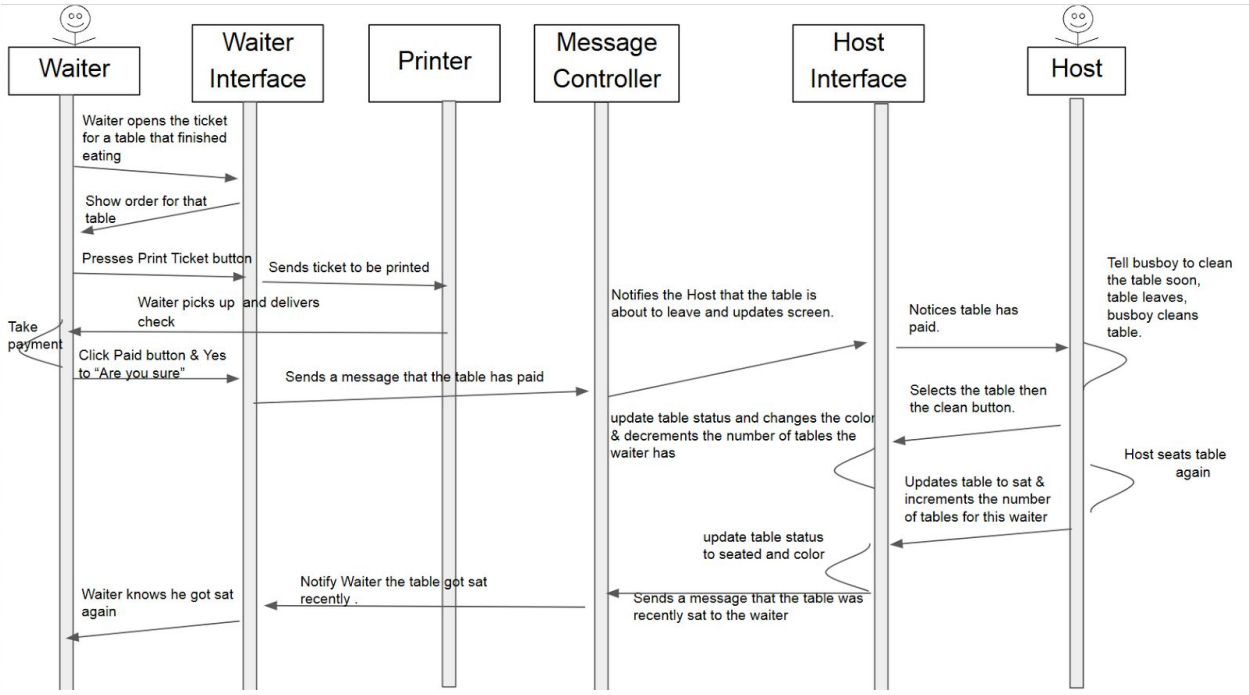


Figure UC1: System Sequence Diagram

Use Case 2 - Finishing a Ticket

Changes we made to Use Case 2:

- Replaced previous use case of "Low Inventory" to "Finishing a Ticket". Low inventory became a use case for an alternate scenario.
- The use case is now from waiter sending the ticket to the chef to waiter picking up hot food.
- The system sequence diagram shows how the status of the ticket changes (specifies the color that it changes to) as the status of each dish is updated by the chef.
- The "Hot Food" notification is added to this use case as a way of letting the waiter know when the dishes are ready to be served.

Casual Description

This use case describes the processes that the different staff members must go through when closing a ticket. It starts from the waiter sending a ticket, the chef starting a dish, the chef finishing all the dishes, and continues all the way to the waiter receiving a hot food notification.

Fully Dressed Description

Related User Stories	ST-HC-F-1, ST-HC-F-2, ST-HC-F-4, ST-HC-F-5, ST-HC-F-6, ST-HC-F-7, ST-HC-F-8, ST-W-F-8, ST-HC-O-1, ST-HC-O-2, ST-HC-O-3, ST-HC-O-4, ST-HC-O-5, ST-HC-O-6, ST-W-O-3
Initiating Actors	Head Chef

Actor's Goal	To finish a ticket
Participating Actors	Waiters
Pre-Conditions	The waiter must have sent a ticket of an order.
Post-Conditions*	Each time the head chef starts a dish on a ticket, the system should be able to acknowledge it by deducting the correct amount of ingredients that the dish uses from the number listed in the inventory (inventory/order database) and updating the status of the ticket. Once the chef has finished all the dishes, the color and status of the ticket should update to finished as well. The waiter should be updated upon this action.
Flow of Events for Main Success Scenario	1→ Head Chef receives a ticket from the waiter. 1.1← The system shows the new ticket in red on the chef's screen as "Unstarted". 2→ Head Chef clicks on the ticket. 2.1→ The system shows the dishes ordered on that ticket. 3→ Head Chef clicks on a dish to start it. 3.1←The system deducts the corresponding quantity of involved ingredients from the inventory. 3.2←The status of the ticket on the home screen should be "Semi-started" and the color changes. 4→ Head Chef changes the status of each dish on the ticket to "started". 4.1← The status of the ticket on the home screen should be "Started" and the color changes. 5→ Head Chef changes the status of each dish on the ticket to "finished" as they are finished being made. 5.1←The status of the ticket on the home screen should be "Finished" and the color changes to green. 5.2←The system sends an alert to the waiter and places a non-invasive banner under the table stating "Hot Food". → Waiter comes to pick up the food. 6→ Head Chef views the waiter take the food and can delete the ticket. 6.1←The system removes the ticket from the screen. 7→ Waiter sends another ticket.

Flow of Events for Extension (Alternate Scenarios)*	1→ Head Chef receives a ticket from the waiter. 1.1← The system shows the new ticket in red on the chef's screen as "Unstarted". 2→ Head Chef clicks on the ticket. 2.1→ The system shows the dishes ordered on that ticket. 3→ Head Chef clicks on a dish to start it. 3.1←The system deducts the corresponding quantity of involved ingredients from the inventory. A) The threshold is met. 4→ The threshold of low stock is met. 4.1←The system sends an alert in the form of a non-interrupting banner to inform the head chef. 4.1.1→ Chef can view the item(s) and quantity left of the ingredient(s) causing the low stock through a button on their screen. 4.2←The system updates the menu screen to reflect the change visually for the server (most likely by changing the color of the item). 4.3←The system sends an alert in the form of a notification to inform the manager. → Manager can choose whether to notify the servers or not. 5→ Manager chooses to alert the servers of low stock. 5.1←The system sends a message to the server alerting them of the ingredient that is running low. 6→ Server receives the message and can accordingly alert the parties who order dishes which use the corresponding ingredient.

System Sequence Diagram

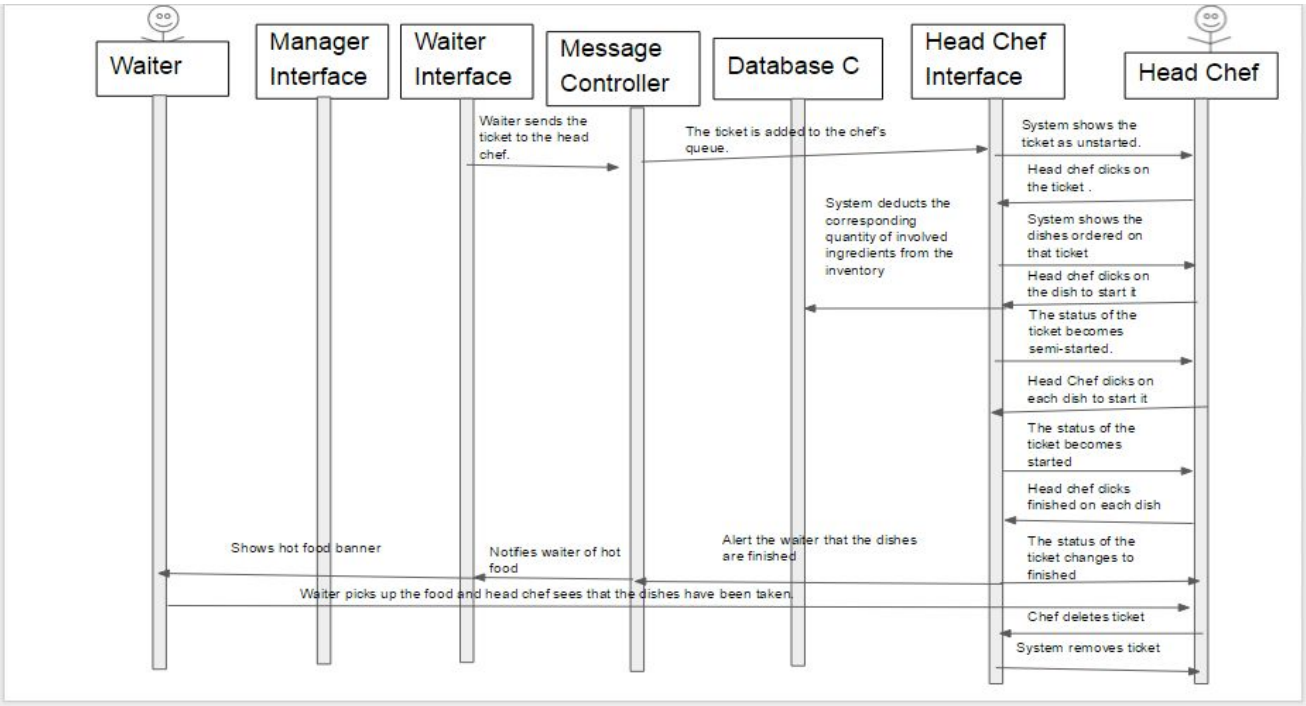


Figure UC2: System Sequence Diagram

Use Case 3 - Placing an Order

Changes we made in Use Case 3:

We changed some of the function names but other than that the use case practically stayed the same. The waiter clicks on the table which open the ticket and then takes their order. The waiter then sends the ticket to the head chef and then the chef will start to prepare the dishes. Once all the dishes on the ticket are prepared the chef will notify the waiter that the food is ready to be served. This happens while also keeping track of the ingredients in the inventory.

Casual Description

The waiter can place an order by opening the ticket for the new table. This is done by clicking on that specific table number. Next the waiter starts adding dishes by clicking the corresponding buttons. After taking the table's order, the waiter sends the ticket to the head chef. When the head chef receives the ticket, he starts preparing the order. The head chef can update the ticket as each dish is started and completed. When all of the dishes on the ticket are completed, the head chef's interface will send a message to the waiter that notifies them that their food is ready to be served. Finally the waiter will go retrieve the dishes and serve them to the customers.

Fully Dressed Description

→ What user does

← what system does in response

Related User Stories	ST-W-F-1, ST-W-F-2, ST-W-F-3, ST-W-F-4, ST-W-F-5, ST-W-F-7, ST-W-F-8, ST-W-F-9, ST-W-F-10, ST-W-O-3, ST-HC-F-1, ST-HC-F-2, ST-HC-F-3, ST-HC-O-5, ST-HC-O-6, ST-HC-O-7, ST-HC-O-8, ST-HC-O-10
Initiating Actor	Waiter
Actor's Goal	To place and complete an order
Participating Actors	Head Chef, Cooks
Pre-Conditions	There must be a button on the waiter's screen to create a new ticket. Also, the customer must place an order.
Post-Conditions	The order has been successfully completed and delivered to the customer.
Flow of Events for Main Success Scenario	1 → Waiter opens new ticket for seated table. 1.1 ← The system shows that the new ticket has been created. 2 → Waiter fills in the information of the ticket. 2.1 ← The system updates the ticket as the information is entered. 3 → Waiter clicks send button. 3.1 ← The system sends a message indirectly to the head chef (first to the message controller who will forward it to the head chef). 3.1 ← The ticket is added to the chef's queue. 3.1.2 ← The ticket gets added to the database. 3.2 ← The system displays the contents of the current ticket on Head Chef's screen. 4 → Head Chef starts cooking the dishes on the current ticket and changes the statuses to started. 4.1 ← The system changes the color of the ticket based on the statuses of the dish on it. 4.2 ← The ingredients needed for the started dish get subtracted from the inventory in the database. 5 → Head Chef updates the status of each dish to finished. 5.1 ← The system changes the color of the finished ticket as all dishes are finished. 5.2 ← The system sends the notification "hot food" to the waiter when all the dishes for their ticket is done. This is an indirect message sent first to

	the message controller who then sends it to the waiter. 6 → Waiter serves the dishes to the table.
Flow of Events for Extension (Alternate Scenarios)	1 → Waiter opens new ticket for seated table. 1.1 ← The system shows that the new ticket has been created. 2 → Waiter fills in the information of the ticket. 2.1 ← The system updates the ticket as the information is entered. A) Waiter changes order. 3 → Waiter clicks on the table to reopen the order. 3.1 ← The system shows the current order and allows more items to be added. 4→ Waiter clicks on items he wishes to add and hits send. 4.1 ← The system confirms and updates the changes in the order by sending a message to the head chef (through the message controller) that will update the head chef's ticket for that table. Progression continues as Main Success Scenario. B) Waiter calls manager for issue. 3 → Waiter notifies manager by pressing a button. 3.1 ← The system sends a message to the message controller who forwards it to the manager. 4 → Manager sees the notification and resolves issue. Progression continues as Main Success Scenario. C) Waiter accidentally presses wrong button. 3 → Waiter clicks the item he just added and then clicks remove button. 3.1 ← The system takes off the item that was highlighted from the ticket. Progression continues as Main Success Scenario. D) Customer is dissatisfied with order. 3 → Waiter notifies manager by pressing a button. 3.1 ← The system sends a notification to the manager about the issue. 4 → Manager resolves the issue with the customer. Progression continues as Main Success Scenario. E) Waiter drops the food. 3 → Waiter apologizes and notifies manager by pressing a button.

	3.1 ← The system sends a notification to the manager about the issue through a message (that is sent indirectly to the message controller that forwards it to the manager). 4 → Manager gets the notification and resolves the issue. 5 → Waiter resends the food that was dropped to Head Chef with a message. 5.1 → The system sends the order to Head Chef by using the message controller. Progression continues as Main Success Scenario.

System Sequence Diagram

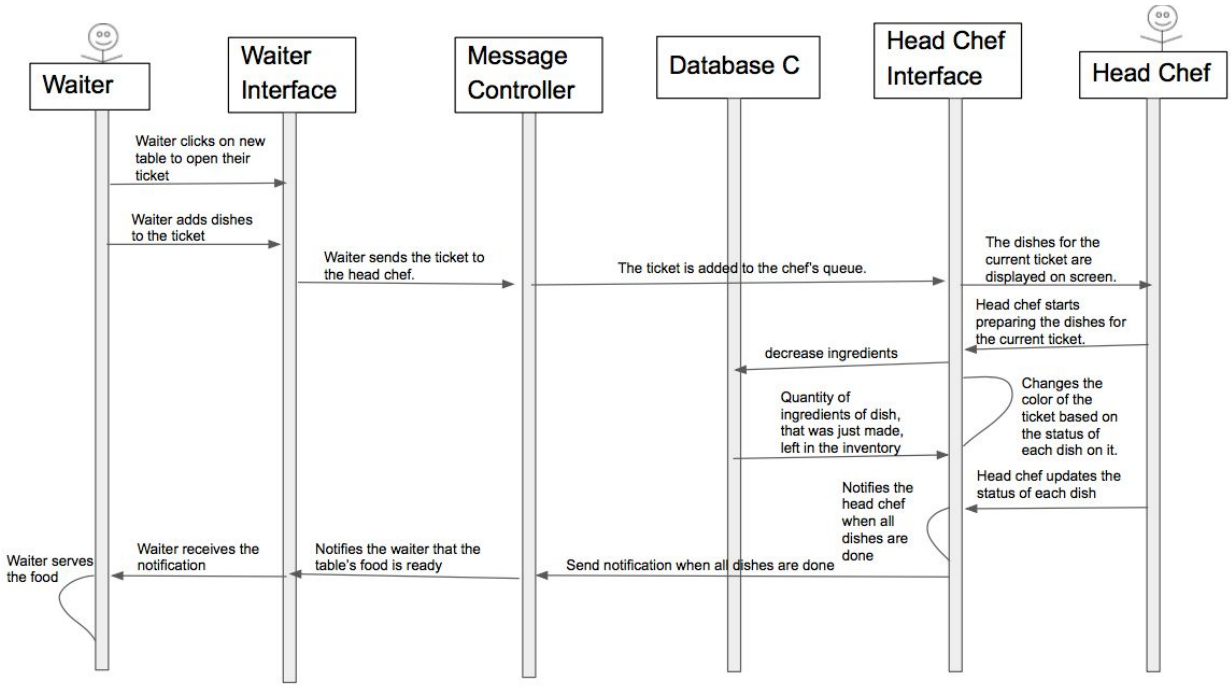


Figure UC3: System Sequence Diagram

Traceability Matrix

	Use Case 1 - Closing a Table	Use Case 2 - Finishing a Ticket	Use Case 3 - Placing An Order
ST-GE-F-1			
ST-GE-F-2			
ST-GE-O-1			
ST-GE-O-2			

ST-GE-O-3			
ST-GE-O-4			
ST-HC-F-1			
ST-HC-F-2			
ST-HC-F-3			
ST-HC-F-4			
ST-HC-F-5			
ST-HC-F-6			
ST-HC-F-7			
ST-HC-F-8			
ST-HC-O-1			
ST-HC-O-2			
ST-HC-O-3			
ST-HC-O-4			
ST-HC-O-5			
ST-HC-O-6			
ST-H-F-1			
ST-H-F-2			
ST-H-F-3			
ST-H-F-4			
ST-H-F-5			
ST-H-O-1			
ST-H-O-2			
ST-H-O-3			
ST-H-O-4			
ST-H-O-5			
ST-M-F-1			
ST-M-F-2			
ST-M-F-3			
ST-M-F-4			
ST-M-O-1			
ST-M-O-2			
ST-M-O-3			
ST-W-F-1			

ST-W-F-2			
ST-W-F-3			
ST-W-F-4			
ST-W-F-5			
ST-W-F-6			
ST-W-F-7			
ST-W-F-8			
ST-W-F-9			
ST-W-F-10			
ST-W-F-11			
ST-W-F-12			
ST-W-O-1			
ST-W-O-2			
ST-W-O-3			
ST-W-O-4			
ST-W-O-5			
ST-O-F-1			
ST-O-F-2			
ST-O-F-3			
ST-O-F-4			
ST-O-F-1			
ST-O-O-2			
ST-O-O-3			

The traceability matrix above illustrates the relationship between use cases their requirements. This is important because this way we can see how the two corresponding concepts overlap and map to each other. As inferred from the chart above, there are some intriguing patterns between the specific use cases and requirements.

User Effort Estimation- Estimation with Use Case Points

PF→ productivity factor =assume 28 hours per use case point

Use Case Weights

Use Case	Description	Category	Weight

Closing a Table (UC-1)	Simple user interface. The main success scenario takes 8 steps. 2 Participating actors.	Average	10
Finishing a Ticket (UC-2)	Simple user interface. The main success scenario takes 5 steps. 2 Participating actors.	Average	10
Placing an Order (UC-3)	Simple user interface. The main success scenario takes 16 steps. 3 Participating actors.	Complex	15

Total= 35

UUCP= 2(Average) + 1(Complex) =35

Technical Complexity Factors

Technical Factor	Description	Weight	Perceived Complexity	Calculated Factor
T1	Distributed System	2	4	8
T2	User expects fast performance	1	2	2
T3	End user efficiency	1	2	2
T4	Complex internal processing with socket/ message controller	1	3	3
T5	No need for reusability	1	0	0
T6	Easy to install	0.5	2	1
T7	User experience is extremely important	0.5	2	1
T8	Some portability concerns if they want to expand to tablets	2	1	2
T9	Easy to change minimally required	1	1	1

T10	Concurrent use required	1	3	3
T11	Security is a concern	1	3	3
T12	No direct access for third party, only owner will have external access	1	0	0
T13	Basic training required to understand interface and usage	1	1	1

Total=27

TCF=0.6+0.01(27)=0.87

Environmental Complexity Factors

Environmental Factor	Description	Weight	Perceived Impact	Calculated Factor
E1	Beginner with UML- Based development	1.5	1	1.5
E2	Familiar with application problem	0.5	2	1
E3	Good knowledge of Object Oriented approach	1	2	2
E4	Beginner lead analyst	0.5	1	0.5
E5	Motivated, some team members occasionally slacking	1	4	4
E6	Stable requirements expected	2	5	10
E7	No part time staff involved	-1	0	0
E8	Programming language of average difficulty (Java)	-1	4	-4

Total=15

ECF=1.4-0.03(15)=0.95

Use case points→ UCP= UUCP x TCF x ECF=35 x 0.87 x0.95=28.9275= 28 Use case points

Duration=UCP x PF= 28 x 28=784 hours

Domain Analysis

Changes in Domain Model Analysis

- We did not include a domain model diagram in our last report so the whole diagram is new as well as the key.

Domain Model Diagram

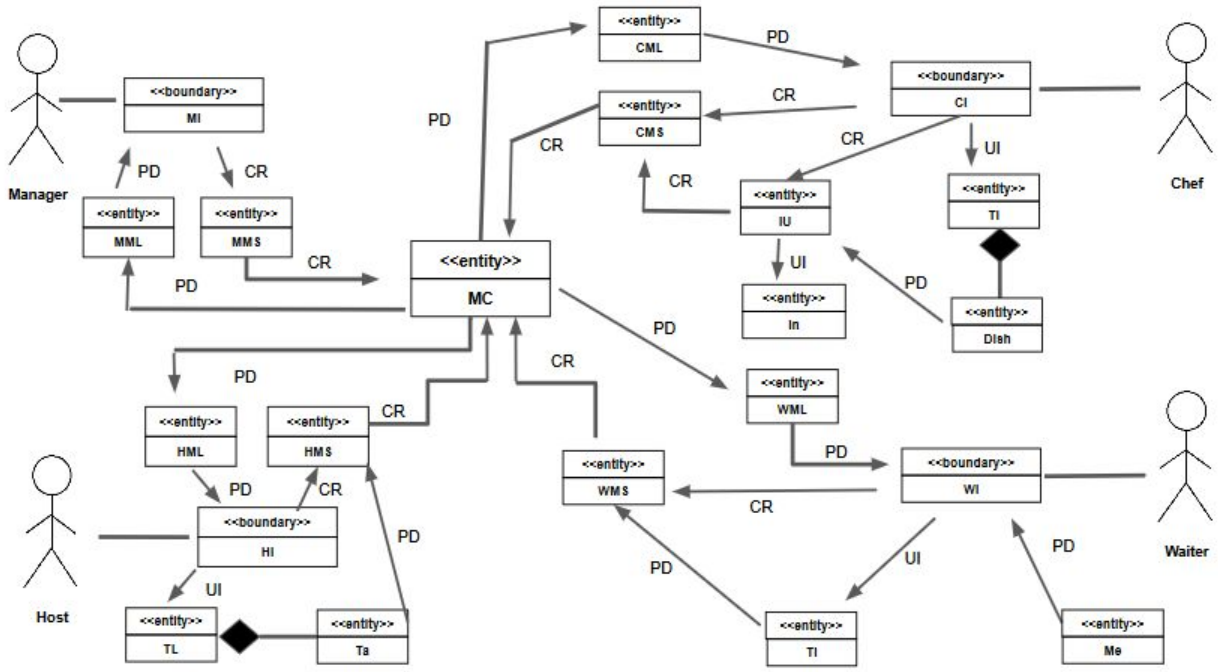


Figure DMD: Domain Analysis Diagram

Key for Figure DMD:

On the arrows:

- PD = Provide Data
- CR = Convey Request
- UI = Update Info

Boundaries:

- MI, CI, HI, WI = Manager, Chef, Host, Waiter Interface

Entities:

- MML, CML, HML, WML = Manager, Chef, Host, Waiter Message Listener
- MMS, CMS, HMS, WMS = Manager, Chef, Host, Waiter Message Sender
- MC = Message Controller
- TL = TableList
- Ta = Table
- Ti = Ticket
- Me = Menu
- Di = Dish
- In = Ingredient
- IU = Inventory Updater

Concept Definitions

Changes we made in Concept Definitions

- Reorganized the chart to put all of the containers first then the doers.
- Removed all of the employee even listeners as all events that happen shall be handled by the employee interfaces.
- Took out the low inventory checker since he was no longer needed. Now the inventory updater will perform the threshold check when it updates an ingredients inventory.
- Took out concepts that dealt exclusively with sending or receiving tickets. A ticket can be sent as a message and therefore does not need its own special functions or concepts to handle transmitting it.

Responsibility Description	**Type**	**Concept**
R-01: Container for the multiple dishes of the ticket. Contains the table number, price, status, and number of the ticket. It also includes the waiter's name and ID.	K	Ticket(Ti)
R-02: Holds the information for the table, including the maximum occupancy, the time the table was last sat, the status of the table, and the name of the waiter who is serving it.	K	Table (Ta)
R-03: Holds all of the tables in the restaurant. It's a list of the tables that allows you to search for a specific table using the table number.	K	TableList (TL)
R-04: Holds each ingredient's data. This includes the amount of the ingredient left in the inventory, the units the ingredient is measured in, the name of the ingredient and the threshold limit for that ingredient.	K	Ingredient (In)
R-05: This container is used to hold the ingredients of one dish on the menu. It will hold the name and amount of ingredients used by this dish. It also holds the name of the dish and the price of the dish.	K	DishData (DD)
R-06: This container holds the dishes name, price, and status. The status can be unstarted, started, or finished which will be determined by the chef.	K	Dish (D)
R-07: This container holds all the dishes on that the restaurant offers to customers. It is organized first by category of dish (like entree, dessert, appetizer) and then	K	Menu (Me)

inside that first organization you will find a list of all the dishes (as dish concepts) that pertain to that category,		
R-08: The container for messages used for communication between subsystems. Holds the sender and receiver's information as well as the content of the message.	K	Message(Mes)
R-09: Container used to represent an employee. Contains the employee's name, unique id, and position at the restaurant. It also holds whether the employee is currently logged in.	K	Employee(E)
R-10: Listens for any messages sent to this waiter interface and calls the appropriate function for handling that message.	D	WaiterMessageListener (WML)
R-11: Sends messages that waiter interface generated to the message controller to be forwarded to the corresponding employee's tablet. This message could be a ticket sent to the chef.	D	WaiterMessageSender (WMS)
R-12: Listens for any waiter action (pressing a button) and calls the appropriate function which may update tickets or send messages. Waiter Interface also acts as the controller of the waiter application, it decides what screen to show and holds the data needed to make the screen.	D	WaiterInterface (WI)
R-13: Listens for any messages sent to this host interface and calls the appropriate function for handling that message.	D	HostMessageListener (HML)
R-14: Sends messages that host interface generated to the message controller to be forwarded to the corresponding employee's tablet.	D	HostMessageSender (HMS)
R-15: Listens for any host action (pressing a button) and calls the appropriate function which may update a table status or send a message. Host Interface also acts as the controller of the host application, it decides what to show on the screen and holds the data needed to make the screen.	D	HostInterface (HI)
R-16: Listens for any messages sent to this chef interface and calls the appropriate function for handling that	D	ChefMessageListener (CML)

message. These messages could include tickets from the waiters.		
R-17: Sends messages that chef interface generated to the message controller to be forwarded to the corresponding employee's tablet.	D	ChefMessageSender (CMS)
R-18: Listens for any chef action (pressing a button) and calls the appropriate function which may update a ticket status or send a message. Chef Interface also acts as the controller of the chef application, it decides what to show on the screen and holds the data needed to make the screen.	D	ChefInterface (CI)
R-19: Listens for any messages sent to this manager interface and calls the appropriate function for handling that message. These messages will be notifications from employees who need the manager's assistance.	D	ManagerMessageListener (MML)
R-20: Sends messages that manager interface generated to the message controller to be forwarded to the corresponding employee's tablet.	D	ManagerMessageSender (MMS)
R-21: Listens for any manager action (pressing a button) and calls the appropriate function which may send a message. Manager Interface also acts as the controller of the manager application, it decides what to show on the screen and holds the data needed to make the screen.	D	ManagerInterface (MI)
R-22: Decrement the amount of ingredients that were used for a dish that was recently started upon receival of a starting dish message from the chef. It will also check if the threshold has been met and if so it will send messages to the appropriate systems to update them of this change.	D	InventoryUpdater (IU)

Association Definitions

Changes we made in Association Definitions:

- A ticket is now sent through the Waiter Message Sender as a message.
- During the event of low inventory, the Inventory Updater will be the one who checks the threshold. Then it will generate a message for the Chef Message Sender to send to the Message Controller. Then the Message Controller will be able to forward it to the manager to notify them.

Concept Pair	Association Description	Association Name

Waiter, Host, Manager, Chef Message Sender⇔ Message	Message provides a structure for the Message Senders to use when conveying their information to the message controller.	Convey information
Waiter, Host, Manager, Chef MessageSender⇔ Message Controller	Message Controller receives a message from employee message sender and sends it to corresponding subsystem	Convey Request
Message Controller ⇔ Waiter, Host, Manager, Chef MessageListener	Message Controller forwards message that it previously received to the employee listener	Provides Data
Waiter, Host, Manager, Chef Message Listener ⇔Waiter, Host, Manager, Chef Interface	An employee's message listener will provide data from the messages it receives to the employee's interface so that the interface can display this data to the user.	Provides Data
Waiter, Host, Manager, Chef Interface ⇔ Waiter, Host, Manager, Chef MessageSender	An employee's interface will generate new messages depending on user interaction and tell the employee message sender to send them.	Convey Request
Host Interface ⇔ Table List	When the host updates a table's status on the screen the interface will update that table in the table list.	Update Information
Table ⇔ Host Message Sender	Table container will provide the host message sender with the information needed to send the message to the correct employee. (Used when sending a recently sat notification to the waiter).	Provides Data

Menu ⇔ Waiter Interface	Menu provides the information about all dishes that the restuarant serves to the waiter interface.	Provides Data
Waiter Interface ⇔ Ticket	Waiter Interface will update a ticket's data when placing an order.	Updates Information
Ticket ⇔ Waiter Message Sender	Ticket will provide the Waiter message sender with the information it needs to send a message. (Used when waiter is ringing in an order to send to the chef).	Provides Data
Chef Interface ⇔ Ticket	Chef interface will update a dish's status on a ticket before he starts preparing the dish.	Update Information
Chef Interface ⇔ Inventory Updater	When the chef starts making a new dish it will tell the inventory updater to decrement the ingredients of that dish.	Convey Request
Dish ⇔ Inventory Updater	When a dish is started by a chef the inventory updater will use the dish to figure out what ingredients it should decrement.	Provides Data
Inventory Updater ⇔ Ingredient	Inventory Updater will decrement the amount of that ingredient in the inventory. It will then check if the amount of that ingredient has dropped passed the threshold.	Update Information
Inventory Updater ⇔ Chef Message Sender	The Inventory Updater will generate a low inventory	Convey Request

	message to send out and tell the Chef's message sender to send it.	

Attribute Definitions

Changes we made in Attribute Definitions:

- Ticket now includes waiter's name and ID.
- Table now includes whether it is a booth or a table.
- Added Ingredient, Dish, Employee, and Menu concept.
- Rewrote each employee's message sender, message listener, and interface description in a more clear and concise way.
- Changed Host Interface to separate table's by their status.
- Changed Chef Interface to separate ticket by their status.
- Merged inventory updater and low inventory checker.

Concept	Attributes	Attribute Description
Ticket(Ti)	Price	Price of the ticket
	Status	Updates according to status of each dish on it
	List of Dishes	Items in each dish
	Table Number	The table that ordered the ticket
	Waiter's Name	Name of waiter who placed order
	Waiter's ID	ID of waiter who placed order
Table (Ta)	Max Occupancy	Maximum amount of people that can be seated at the table.
	Status	Tells you the status of the table (seated, paid, ready)
	Table Number	The table's number that will be used to identify it.
	Waiter Name	Name of the waiter who is serving the table

	BoothOrTable	The type of table this table is, can be a booth or a table.
Table List (TL)	List Of Tables	Holds a table object for each table in the restaurant.
Ingredient(In)	Name	Name of ingredient.
	Amount	Amount of ingredient you have in the inventory
	Unit of Amount	The unit that the amount is measured in. (cups, oz, lbs)
	Threshold	Lower limit on amount of ingredient, when passed it will signify low inventory.
Dish (D)	Name	Dish's name
	Price	Price of Dish
	Status	Status of Dish (unstarted, started, finished)
Menu (Me)	List Of Apps	List of appetizers that the restaurant offers. Each app is type Dish.
	List Of Entrees	List of entrees that the restaurant offers. Each entree is type Dish.
	List Of Desserts	List of desserts that the restaurant offers. Each dessert is type Dish.
	List of Drinks	List of drinks that the restaurant offers. Each drink is type Dish.
Message (Mes)	Sender	Who sent the message

	Recipients	Who is receiving the message
	Data	Contents of message
Employee (E)	Name	Name of employee
	Position	Position of employee. IE: waiter, chef, manager, host
	ID	Unique number used to log in
Waiter,Host,Chef,Manager MessageListener (XML) (X = W or H or C or M)	List of Messages	List of messages sent to this tablet.
	Decoder	Decode a message and perform the appropriate action. IE for the WML the decoder will read a recently sat message and tell the waiter interface to make a notification on the screen.
Waiter,Host,Chef,Manager MessageSender (XMS) (X = W or H or C or M)	Pending Messages	List of messages that need to be sent out.
	Address of Message Controller	This will be where the sender sends the messages too. From there the message controller can forward the message to the correct listener.
Waiter Interface (WI)	List Of Tickets	List of all the tickets the waiter currently has. Each ticket will represent a table that the waiter is taking care of.
	Current Ticket	The current ticket that the waiter has open on the screen.
	Menu	List of all the names and prices of dishes on the menu.

Host Interface (HI)	Ready Tables	List of all the tables in the restaurant that are ready to be seated.
	Seated Tables	List of all the tables in the restaurant that are seated.
	Paid Tables	List of all the tables in the restaurant that have paid and need to be cleaned.
	List of Waiters	List of all the waiters who working.
Chef Interface (CI)	Current Ticket	The ticket that the chef has opened on his screen and is updating dish statuses on.
	Unstarted Tickets	List of tickets that have all their dishes unstarted.
	Semi started Tickets	List of tickets that have at least one dishe unstarted.
	Started Tickets	List of tickets that have all their dishes started or finished.
	Finished Tickets	List of tickets that have all their dishes finished.
Inventory Updater (IU)	MenuDishData	List of all the dishes, their ingredients, and the amount of ingredients the dish uses.
	Inventory	List of all the ingredients of the kitchen and how much of each ingredient is in the kitchen.
	Thresholds	The minimum amount the inventory can reach for each ingredient before a notification must be sent.

	Address of Chef Listener	This will tell the IU where to send the low inventory notification.

System Operation Contracts

Use Case 1

Changes we made in Use Case 1:

- Instead of changing the color of the ticket on the waiter screen we actually remove it from the list as they should no longer be able to access it again since it is paid.

Name	Closing a Table
Responsibilities	Print out the check to give to the table, and take the payment for that check so the party can leave. Notify the host that table has paid.
Use Cases	UC-1
Exceptions	None
Pre-Conditions	Check is viewable on-screen and there is a print check button on the bottom of the screen that will print the check at the printer.
Post-Conditions	The table should no longer be shown on the waiter's screen. The Host's screen should be updated to include a paid status for the departing table.

Use Case 2

Changes in Use Case 2:

Instead of Low Inventory, we discussed the actions involved in Finishing a Ticket. This changed all sections of the System Operation Contracts.

Name	Finishing a Ticket
Responsibilities	Update the status of the ticket as each dish is started. When all the dishes are finished, send notification "Hot Food" to the waiter so that he could come and pick it up.
Use Cases	UC-2
Exceptions	None

Pre-Conditions	The dish names which are buttons are viewable on-screen and will allow the modification of the status of the dish. There should also be a delete button to allow the chef to delete the ticket.
Post-Conditions	The ticket on the chef's screen should change color according to the status updates of the dishes. When all the dishes are finished, the waiter's screen should be updated to include a "Hot Food" Banner. If the ticket is not needed, the ticket should no longer be shown on the chef's screen.

Use Case 3

Changes in Use Case 3:

- Waiter doesn't need to have a button to create a ticket. A new ticket should already exist once they get sat a table so all they have to do is click on that specific table.

Name	Placing an Order
Responsibilities	To complete an order.
Use Cases	UC-3
Exceptions	None
Pre-Conditions	The Waiter's screen must show the tables that they have been sat so they can open up the tickets for each table. Also, the customer must place an order.
Post-Conditions	The order has been successfully completed and delivered to the customer.

Traceability Matrix

Use Case	Domain Concepts										
	Ti	Ta	TL	In	DD	D	Me	Mes	E	WML	WMS
Use Case 1	X	X						X	X	X	X
Use Case 2	X	X		X	X	X	X	X	X	X	X
Use Case 3	X	X						X	X	X	X

Use Case	Domain Concepts (continued)										
	WI	HML	HMS	HI	CML	CMS	CI	MML	MMS	MI	IU
Use Case 1	X	X		X							
Use Case 2	X				X	X	X				X
Use Case 3	X				X	X	X				X

The traceability matrix above depicts the relationship between the domain concepts and use cases. Each 'X' indicates that the domain concept is present in each respective use case. Domain concepts can be mapped to multiple use cases to show how they are interrelated. Please also note the traceability matrix above was split into two to be visualized more easily.

Mathematical Model

Changes we made in Mathematical Modelling:

- Replaced the host event listener with host interface. The host interface will listen for buttons clicked by the host and also handle requests from the host message listener.
- Added special functions to handle a host pressing the cleaned button and a host pressing the seat button.
- Since we are sending Recently Sat messages to waiters, we needed to add the listOfWaiters so the host interface can get the waiter's id from the waiter's name and pass the id to the message controller so it can send it to the right waiter.

Generating Host's Table Queue:

Algorithm for generating and updating the host's two table lists. The host screen will show ready tables (tables that are open) in one list and not ready tables (tables that are seated or dirty) in another, both on one screen. Below is pseudocode for the algorithm we will use for ordering and organizing these table lists.

Pseudo Code:

```
At start of day:
        allTables = getListofTableNumbers();
        readyTables = new TableList();
        notReadyTables = new TableList();
        for (each table in allTables)
                table.status=ready;
                readyTables .add(table);
        listOfWaiters = getWaitersLoggedInFromDBA();
```

Throughout the day: The host interface will listen for the host clicking the screen or an on check message from a waiter.

The following methods will be in the host interface:

```
//This will listen from messages from the message controller
hostMessageListener(Message m){
        //If the waiter sent you a message- means they have paid
        if(m.senderInfo.pos == 'w'){
                String tableNumber = m.content;
```

```
                Int tN = Integer.parseInt(tableNumber);
                Table t =notReadyTables.get(tN);
                t.changeStatus('u');
        }
        Else if(m.senderInfo.pos == 'm'){
                displayNotifcation(m.content);
        }
        redrawScreen();
}

cleanedButtonClicked(int tabNum ){
        Table t =notReadyTables.get(tabNum );
        t.changeStatus('r');
        notReadyTables.remove(tabNum );
        readyTables.add(t);
}

seatedButtonClicked(int tabNum, String waiterName ){
        Table t =readyTables.get(tabNum );
        t.changeStatus('s');
        t.changeWaiterName(waiterName);
        Long waiterID = listOfWaiters.get(waiterName);
        t.changeWaiterID(waiterID);
        readyTables.remove(tabNum );
        notReadyTables.add(t);
        sendRecentlySatNotification(waiterID, tabNum);
}
```

Interaction Diagrams

Use Case 1 - Closing a Table

Changes we made in Use Case 1:

- Added the actual names of the functions into the interaction diagram.
- Changed and took out Database B messages about seating a table. This is no longer needed as the Host Interface will hold all of this information. Database B will hold the list of tables but not their active statuses.
- Changed the on check status to a paid status. On check meant that a table has not paid yet but paid means that they have already paid and they will be ready to leave soon.
- Added a list of waiters to the host screen which holds not only the names of waiters that can be sat but also the amount of active tables each waiter currently has. We had to make sure to update this when statuses of tables were changed.

- Implemented the Command Pattern Design principle. The message controller acts as the command in the model.
- Explained the socket communication more thoroughly.

Observe Figure UC1 below.

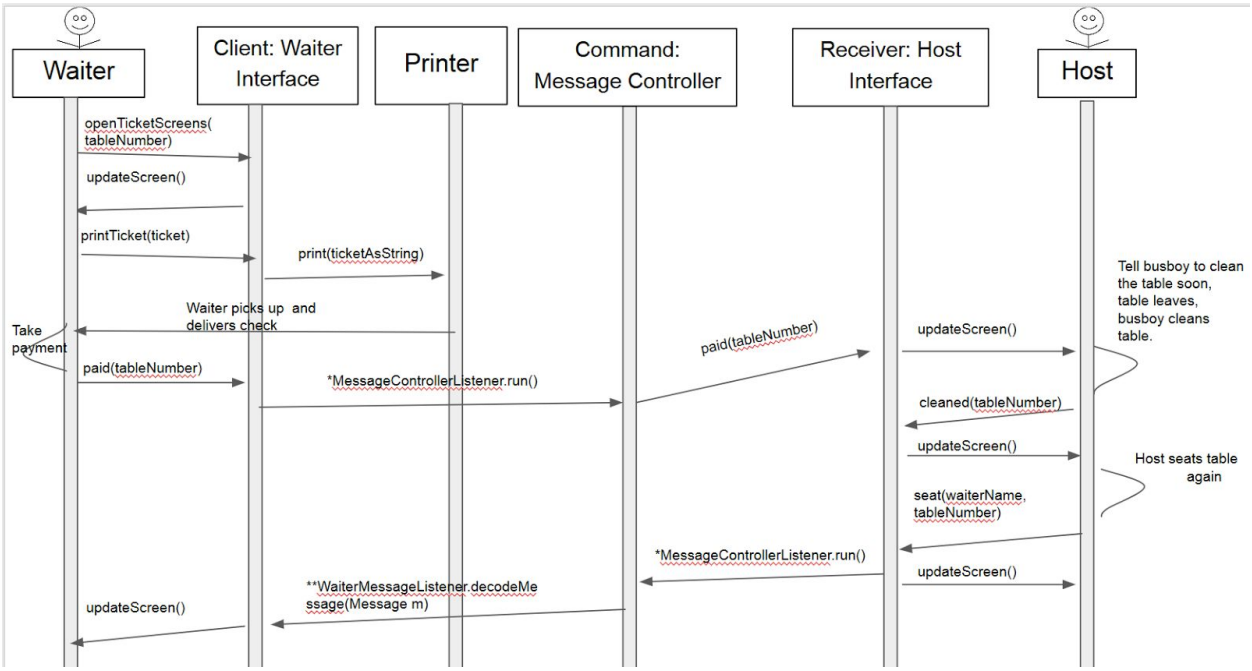


Figure UC1: Closing a Table

Description:

The waiter can print a ticket by first opening that table's order. This is done by clicking the button with that table number on it on the waiter's screen. Next the waiter will click the print button to send the ticket to the printer to be printed. From there the waiter will bring the printed ticket to the table, collect the payment and press the paid button on his screen. This will update his screen and also send a message to the Host. Sending the message to the host is done through sockets (see below). The notification will alert the host about the table that is on check. After that table is cleaned, the Host will click the status of that table to update it to ready. Then after he sits it again, he will update it to seated which will send a message to the corresponding waiter.

Comments on Message Controller:

Since all of the tablet's connect to the Message Controller to send and receive messages, there is actually two threads made each time a tablet connects with the message controller. One thread for sending messages to that tablet and one for listening for messages from that tablet. Each sender thread has a list of pending messages. When any message controller listener gets a message that should be sent to the host tablet then it will add that message to the pending

messages of the message controller sender for that host. Since this is all impemented using sockets, there is really no function to call when sending a message to the Message Controller because it is done by writing to a socket and then reading from a socket. In the Message Controller Listener thread, this is done in the run() function (hence the *MessageControllerLister.run()).
**WaiterMessageListener.decodeMessage(): Much like the Message Controller the Waiter Interface is also multithreaded. One thread will be used to listen to incoming messages from the Message Controller. This thread is defined in class WaiterMessageListener. It is still apart of the Waiter Interface but since the decodeMessage() is not defined in WaiterInterface.java but in WaiterMessageListener.java the arrow specifies the WaiterMessageListener prefix. The decode message will create a ticket to add to the Waiter's List of tickets and mark it as Recently Sat. This is all done inside the decode message function.

Design Principles and Design Patterns

1. The Expert Doer principle is implemented by the:
    a. Printer. The printer knows how to print a ticket properly given a ticket and so handles this task.
    b. Message Controller. The message controller is able to decode a message in order it to forward it to the correct destination. Instead of having each device having an additional task of sending a message to a specific device, we gave this task to one message controller.
    c. Database B. Database B holds the list of table numbers and their information. It does not modify the data on its own but just holds it.
2. The High Cohesion principle is implemented by the:
    a. Printer, Message Controller, and Database B. This because as an expert doer, having only one task allows the concept to keep the amount of computations to a minimum.
    b. Host Interface. The host interface uses the high cohesion principle because it only listens for an event (message received or button pressed) and handles both of these the same way computationally by changing the status of a table.
3. The Low Coupling principle is implemented by the:
    a. Message Controller. It reduces the amount of overhead for each of the other devices since they no longer need their own message controller system. The message controller lowers other subsystem coupling.
    b. Host Interface. This is a direct path to the Database B which doesn't involve the message controller.
4. Command Pattern design is implemented:
    a. Message Controller: It accesses the API between two clients or a client and a server. The message controller knows the sender's request and also keeps track of

the receivers so that it can forward a message to the correct destination. We don't do anything reversible because when a table has paid the bill and left, there should be no reason to undo the change from seated to paid until another new party has been seated. We also don't store the command history because we feel that it is not as necessary to save the messages as it is to save the tickets which is done in Database C.

Use Case 2 - Finishing a Ticket

Changes we made in Use Case 2

- Replaced our previous Use Case idea (Low Inventory) with Finishing a Ticket, upon the feedback provided.
- The interaction diagram is also changed now to reflect this new case.
- Descriptions of actions were replaced with functions existing in code.

Observe Figure UC2 below.

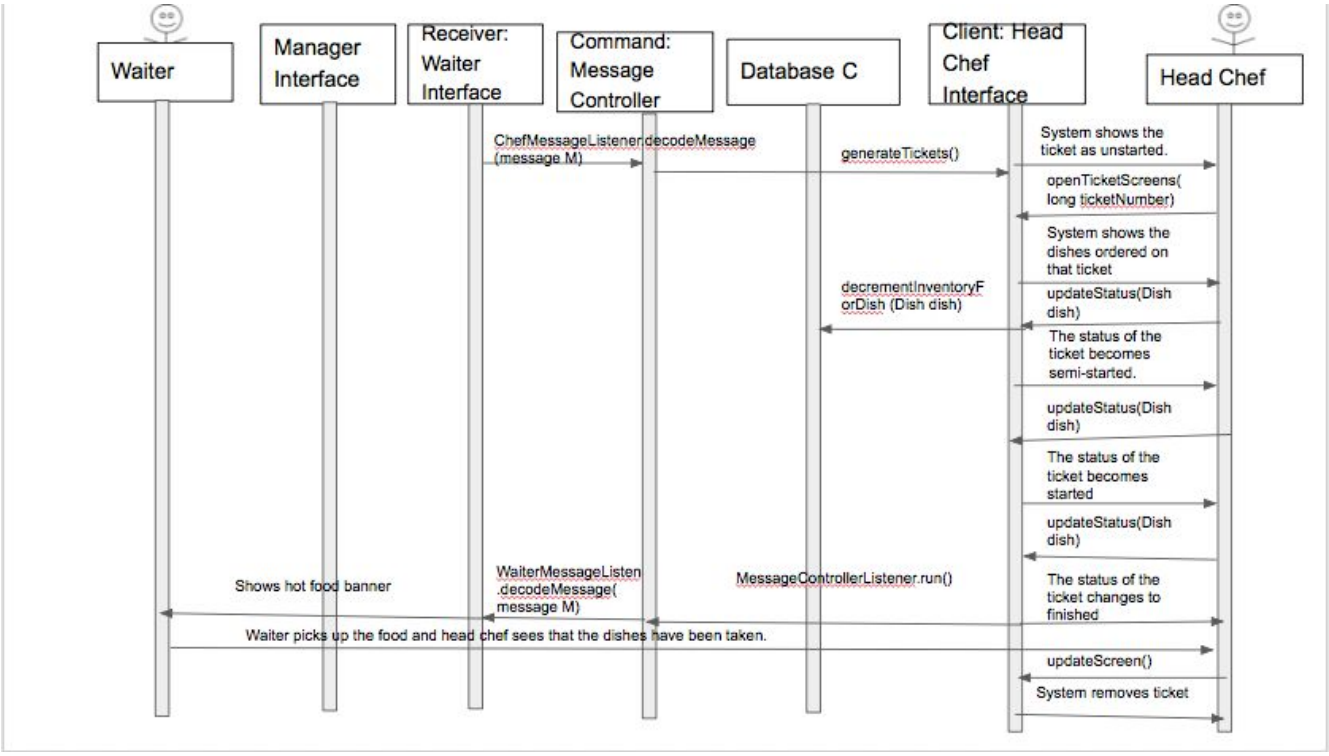


Figure UC2: Finishing a Ticket

Description

This diagram shows the functions used when Finishing a Ticket. It shows the actions from when the waiter sends a ticket (orders) to the head chef to when they come to pick it up. In between is the focus of the use case— how the head chef interacts with the ticket and starts/finishes dishes. The diagram also shows a major component needed for communication

between the waiter and the head chef: the message controller. Tickets and updates are sent through the controller and such correspondence is shown in the diagram.

Design Principles and Design Patterns

1. Expert Doer Design Principle:
    a. Message Controller : The message controller uses the expert doer principle because its main task is to parse and decode the message sent from the interfaces and forward them to the appropriate interface.
    b. Database C: The database uses the principle because it holds all the data on the menu items, and the ingredients.
2. High Cohesion Design Principle:
    a. Database C, Message Controller: The database and the message controller use expert doer principle as stated but they also use the high cohesion principle. Database C uses it periodically when the stock of the ingredients is lowered upon the starting of a dish. The message controller uses it only when the interface chooses to send a message to another interface.
    b. Head Chef Interface: The head chef interface uses the high cohesion principle because it only has to listen for an event which requires modifying the database, such as starting a dish (thus lowering the stock).
3. Low Coupling Design Principle:
    a. Message Controller: The message controller uses low coupling principle because having a central messaging system lowers the amount of responsibilities of the other interfaces in terms of communication with other subsystems.
    b. Head Chef Interface: The head chef interface uses the low coupling principle because it has direct access to database C and can perform actions to modify the data without using the message controller.
4. Command Pattern Design Pattern:
    a. This Use Case uses the Command Pattern Design Pattern. As the Command Pattern describes, the use case separates the parameter preparation from program control (the decision of when to call). The pattern shows Client A → Command → Receiver B. Client A creates parameters for the command (parameter preparation) and Command executes them with Receiver B (program control). In our use case the Client is the Head Chef Interface, the Command is the Message Controller, and the Receiver is the Waiter Interface. This pattern allows the command (the message controller) to have a uniform method signature and allows for easier separation. In our case, this is helpful because the interfaces are no longer dependent on each other, but rather linked through the controller (the command). An example of the separation is when the head chef finishes a ticket and the status is updated. The new status sets up the ticket number as a parameter

and sends it to the message controller. The message controller then interprets the information and accordingly sends to the waiter. These actions are separate (decoupled). This method allows for us to further develop the client, command, or receiver without issues.

Use Case 3 - Placing an Order

Changes we made to Use Case 3:

- Changed void openTicket(int TableNumber) to void openTicketScreens(int tableNumber) between Waiter and Waiter Interface.
- Changed Display Order to Display Ticket between Waiter Interface and Waiter.
- Changed void addDish(Ticket t, Dish d) to void addDishToTicket(Dish dish) between Waiter and Waiter Interface.
- Changed void waiterMessageListener(Message messToWaiter) to void decodeMessage(Message m) between Message Controller and Waiter Interface. Similarly, we also changed void chefMessageListener(Message messToChef) to void decodeMessage(Message m) between Message Controller and Head Chef Interface.
- Changed decreaseStock to decrementInventoryForDish between Head Chef Interface and Database C
- Changed void startDish(Ticket t, Dish d) to updateStatus(Dish dish) between Head Chef and Head Chef Interface.Similarly, we also changed void updateDish(Ticket t, Dish d) to updateStatus(Dish dish) between Head Chef and Head Chef Interface.
- Modified void sendMessage(Message hotFood) to void sendMessage(Message m) between Head Chef Interface and Message Controller

Observe Figure UC3 below.

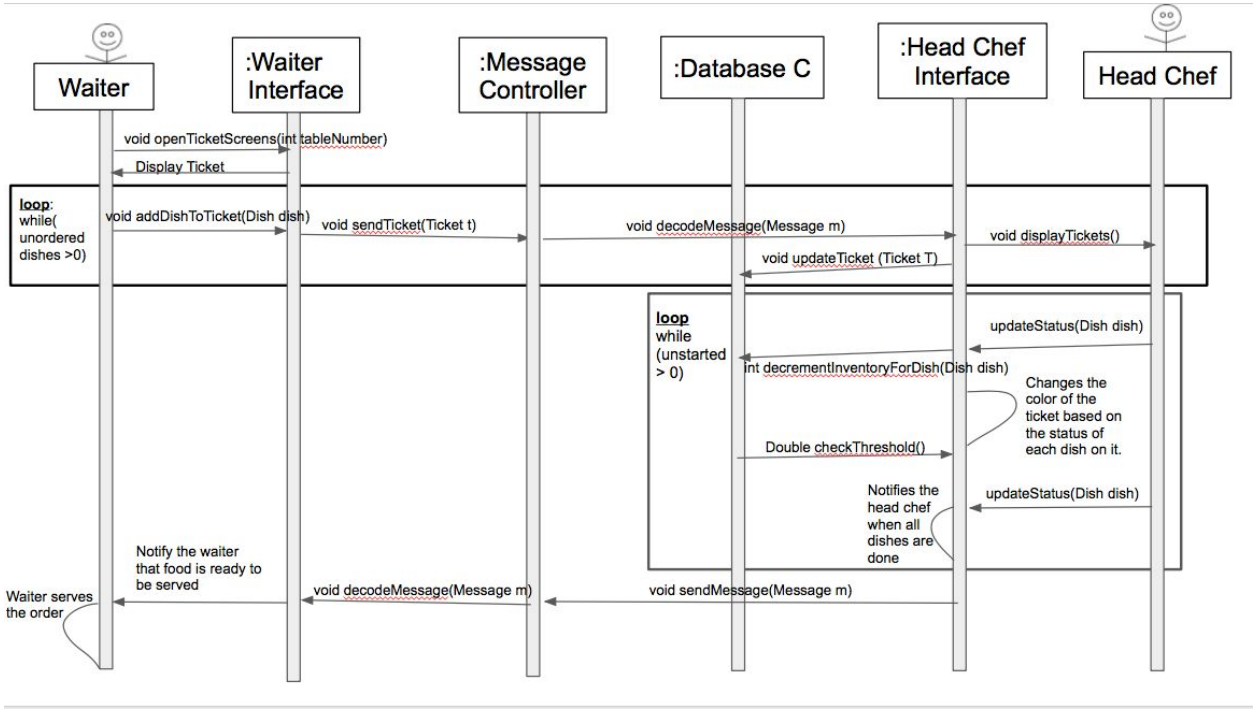


Figure UC3: Placing an Order

Description:

The diagram shows the functions that are used when Placing an Order. It shows the actions from when the waiter first approaches a table to when they serve the table their food. In the middle of that is the focus of this particular use case, how the waiter interacts with the chef to place and serve an order. The diagram also shows a major component needed for communication between the waiter and the head chef: the message controller. Tickets and notifications are sent through this message controller as shown in the diagram.

Design Principles and Design Patterns

1. The Expert Doer principle is implemented by the:
    a. Message Controller. The message controller is able to decode a message in order it to forward it to the correct destination. Instead of having each device having an additional task of sending a message to a specific device, we gave this task to one message controller.
    b. Database C: The database uses the principle because it holds all the data on the menu items, ingredients, and the current orders and has no other function.
2. The High Cohesion principle is implemented by the:
    a. Message Controller and Database C. This because as an expert doer, having only one task allows the concept to keep the amount of computations to a minimum.
    b. Waiter Interface. The waiter interface uses the high cohesion principle because it only interacts with message controller. It has two jobs, one is to listen for a message sent to the waiter or to send a ticket.
3. The Low Coupling principle is implemented by the:
    a. Message Controller. It reduces the amount of overhead for each of the other devices since they no longer need their own message controller system. The message controller lowers other subsystem coupling.
    b. Head Chef Interface: The head chef interface uses the low coupling principle because it has direct access to database C and can perform actions to modify the data without using the message controller.
4. Command Pattern Design Pattern:
    a. Use Case 3 implements the Command Pattern Design Pattern. With this pattern the use case separates the parameter preparation from program control. This specific pattern goes from Client (Waiter Interface) to Command (Message Controller) to Receiver (Head Chef Interface). The client creates parameters for the command (parameter preparation) and the command executes them with the receiver (program control).

Class Diagrams & Interface Specification

Changes we made in Class Diagrams & Interface Specification:

- Deleted functions from Host Interface, Waiter Interface, Head Chef Interface, Message Controller, Manager Interface, DBA Controller.
- Added classes DBB Controller, DBC Controller
- Edited the Class Key Identifier number in the diagram accordingly.
- Edited the placement of classes to accommodate new classes to properly show how they are associated.
- Added functions to Host Interface, Waiter Interface, Head Chef Interface, Message Controller, Manager Interface, DBA Controller, DBB Controller, DBC Controller.

Class Diagram

Observe Figure CD below. A key with functions for each class is expanded underneath. Boxes 0, 1, 2, 8, 14, 15, and 16 represents the SQL tables of data. They have not been implemented as of yet, but will be in the future.

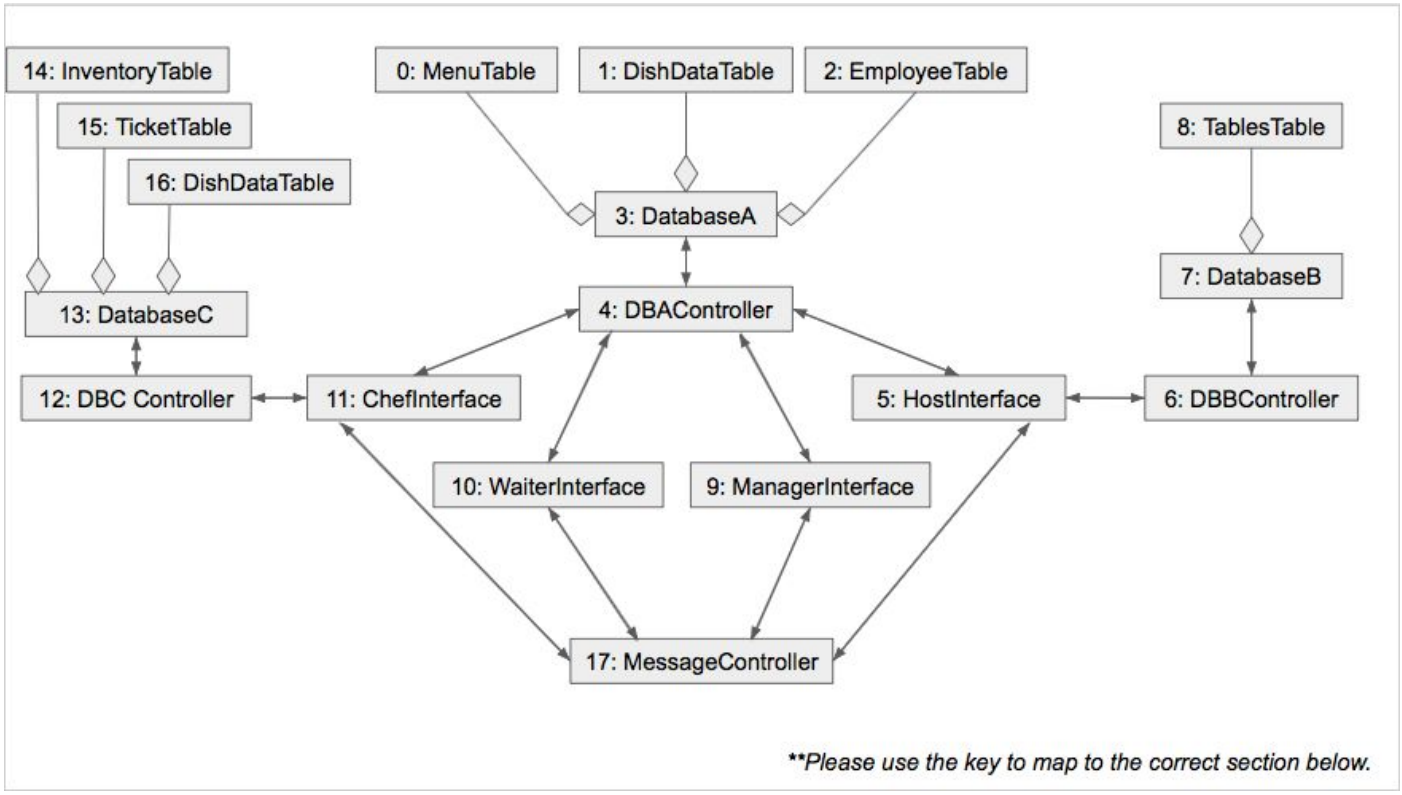


Figure CD: Class Diagram

Class Diagram Key:

0. Menu Table
    a. Attributes:
        i. String dishType
        ii. String dishName
        iii. int dishDataID
1. Dish Data Table
    a. Attributes:
        i. int dishDataID
        ii. String ingredientName
        iii. double ingredientAmount
        iv. String unit
2. Employee Table
    a. Attributes:
        i. long employeeID
        ii. String name
        iii. char position
        iv. String tabAddr
3. Database A
    a. Attributes:
        i. DB databaseA
4. Database A Controller (DBA Controller)
    a. Attributes:
        i. Socket currListener
        ii. Long currentID
        iii. ArrayList<Employee> employeeList
    b. Functions:
        i. Boolean addEmployee(String Name, char position)
        ii. String sendWaiters()
        iii. void loadEmployees()
5. Host Interface
    a. Attributes:
        i. HashMap<Integer,Table> allTables
        ii. ArrayList< Integer> readyTables
        iii. ArrayList< Integer> seatedTables
        iv. ArrayList< Integer> paidTables
        v. HashMap<String, Integer> listOfWaiters
        vi. HashMap<String, Integer> waiterTotalTables
        vii. HashMap<Integer,String> waiterOfTable
        viii. LogInScreen loginPanel
        ix. Long empID
        x. String name
        xi. HostMessageSender sender
        xii. HostTableScreen tableScreen

11. Chef Interface
    a. Attributes:
        i. Long empID
        ii. String name
        iii. Boolean loggedOut
        iv. Long currTicketNumber
        v. HashMap<Long, Ticket> ticketLookup
        vi. ArrayList<Long> ticketQueueUnstarted
        vii. ArrayList<Long> ticketQueuesemiStarted
        viii. ArrayList<Long> ticketQueueStarted
        ix. ArrayList<Long> ticketQueueFinished
        x. LogInScreen loginPanel
    b. Functions:
        i. void chefTicketListener(Ticket ticket)
        ii. void decrementInventoryForDish(Dish dish)
        iii. void changeTicketLocation(char oldstatus, Ticket t)
        iv. void setUpMessageController()
        v. void openTicketScreens(long ticketNumber)
        vi. void backtoMainScreen()
        vii. void logOut()
        viii. void notifyManager()
        ix. void updateScreen()

12. Database C Controller (DBC Controller)
    a. Attributes
        i. HashMap<String, Ingredient> inventory
        ii. HashMap<String, DishData> dishData
        iii. HashMap<Integer,Sender> waiterSenders
        iv. Sender chefSender
        v. Menu menu
    b. Functions
        i. boolean addIngredientToInventory(String ingredientName,Double amountLeft, String unitOfAmount, Double threshold)
        ii. int addDishtoMenu(String type, String dishname, double price)
        iii. void recordTicket(String tick)
        iv. void decrementDish(String dishName)

b. Functions:
   i. boolean loadTables()
   ii. void setUpMessageController()
   iii. void notifyManager()
   iv. void notifyWaiter(Message m)
   v. int seat(String waiterName, int tableNumber)
   vi. void sendSeated(long waiterId, int tableNumber)
   vii. void addNotification(String content)
   viii. void logOut()
   ix. void updateScreen()
   x. void paid(int tableNumber)
   xi. void cleaned(int tableNumber)
   xii. boolean loadWaiters()
   xiii. long getWaiterIdForTable(int tableNumber)

6. Database B Controller (DBB Controller)
   a. Attributes
      i. Socket currListener
      ii. TableList listOfTables
   b. Functions
      i. boolean addTable(int tabNum, int maxOccupancy, char type)
      ii. void loadTablesFromFile()
7. Database B
   a. Attributes:
      i. DB databaseB
8. TablesTable
   a. Attributes:
      i. int tableNumber
      ii. Date timeLastSat
      iii. char status
      iv. int maxOcc
      v. string waiter
9. Manager Interface
   a. Attributes:
      i. LinkedList<Message> listOfMessages
      ii. ManagerrMessageSender sender
      iii. JFrame frame
      iv. Long empID
      v. String name
      vi. Boolean loggedOut
      vii. LogInScreen loginPanel
      viii. ManagerScreen manScreen
   b. Functions:
      i. void logOut()
      ii. void updateScreen()
      v. void sendLowInventoryNotifications(Ingredient i)
      vi. void loadMenuFromFile()
13. Database C
   a. Attributes:
      i. DB databaseC
14. Inventory Table
   a. Attributes:
      i. String ingredientName
      ii. int ingredientID
      iii. double currentAmount
      iv. double threshold
15. Ticket Table
   a. Attributes:
      i. int tableNumber
      ii. char ticketStatus
      iii. long waiterID
      iv. date timeStamp
      v. string dishName
      vi. int dishDataID
      vii. char dishStatus
16. Dish Data Table
   a. Attributes:
      i. int dishDataID
      ii. string ingredientName
      iii. double ingredientAmount
      iv. string unit
17. Message Controller
   a. Attributes
      i. HashMap<Long,MessageControllerSender> waiterOut
      ii. HashMap<Long,MessageControllerSender> hostOut
      iii. HashMap<Long,MessageControllerSender> chefOut
      iv. HashMap<Long,MessageControllerSender> managerOut
   b. Functions
      i. void addWaiterSender(long id,MessageControllerSender sender)
      ii. void addChefSender(long id,MessageControllerSender sender)
      iii. void addHostSender(long id,MessageControllerSender sender)
      iv. void addManagerSender(long id,MessageControllerSender sender)

iii. void setUpMessageController()
iv. void deleteMessage(int index)
v. void sendMassNotification(String content)
vi. void addMessageToList(Message m)

10. Waiter Interface
    a. Attributes:
        i. WaiterMessageSender sender
        ii. DataBaseCSender DBCSender
        iii. JFrame frame
        iv. Long empID
        v. String name
        vi. Boolean loggedOut
        vii. Ticket currTicket
        viii. Menu menu
        ix. HashMap<Integer,Ticket> listOfTickets
        x. WaiterTickListScreen ticketListScreen
        xi. WaiterOneTicketScreen oneTickScreen
    b. Functions:
        i. void openTicketScreens(int tableNumber)
        ii. Boolean loadMenu()
        iii. Void logOut()
        iv. void paid(int tableNumber)
        v. Void notifyManager(Ticket currTicket2)
        vi. Void sendTicket(Ticket t)
        vii. Void addGCorCoupon(double price)
        viii. Boolean addDishToTicket(Dish dish)
        ix. Void removeDishFromTicket(int indexInTicket)
        x. Void addComment(int ind, String com)
        xi. Void backToMainScreen()
        xii. Void addNotification(String content)
        xiii. Void updateScreen()
        xiv. Void setUpMessageController
        xv. Void notifyWaiter(int tableNumber)
        xvi. Void removeLowInventoryDishes(String[] dishes)

v. void removeWaiterSocket(long id)
vi. void removeChefSocket(long id)
vii. void removeHostSocket(long id)
viii. void removeManagerSocket(long id)

Class Descriptions

Database A - Database A is a class that holds the list of employees for the restaurant, along with their information. It also holds the complete menu with every dish's data.

Database A Controller- Database A Controller is a class that manages Database A. It has functions that allow employees to login (logging in means giving the employee id and getting back the position of the employee). It has a special function for when a waiter logs in, this function returns the menu.

DishData- DishData is a class that holds the name of the dish and amount of each ingredient needed for that dish.

Employee- Employee is a class that holds the employee name, id and position.

Database B - Database B is a class that holds the list of tables.

Table- Table is a class that holds a table's status, maximum occupancy, the waiter serving it and the last time it was sat.

Host Interface- Host interface is a class that acts as a controller for Database B and manages what the host sees on the tablet. It listens and sends messages. It handles the actions coming from the host.

Manager Interface - Manager interface is a class that listens and sends messages. It displays it's recently received messages to the manager.

Waiter Interface- Waiter interface is a class that manages what the waiter sees on his tablet. It listens and sends messages. It handles the actions coming from the waiter. It displays the menu to the waiter so they can generate tickets.

Chef Interface - Chef Interface is a class that acts as a controller for Database C, and manages what the chef sees on the tablet. It listens and sends messages. It handles the actions coming from the chef. It displays the queue of tickets.

Database C- Database C is a class that holds the inventory (list of ingredients) and ticket queue. This also audits tickets

Ingredient- Ingredient is a class that holds the name of the ingredient, the current amount of that ingredient in the inventory, the unit of the amount and the threshold for that ingredient.

Ticket- Ticket is a class that holds a list of dishes, the table number of that ticket, the time the ticket was received and the waiter who sent that ticket.

Dish- Dish is a class that holds the name of the dish, DishData for that dish and the status of that dish.

Message Controller- Message Controller is a class that acts as a communication center for all interfaces. It holds the list of messages it needs to send as well as the destination address of each interface

Message - Message is a class that is used by Message Controller and each Interface. It holds the sender, receiver, time it was received and the contents of the message.

Traceability Matrix

	Software Classes															
Domain Concepts	Database A	Database A Controller	Dishdata	Employee	Database B	Table	Host Interface	Manager Interface	Waiter Interface	Chef Interface	Database C	Ingredient	Ticket	Dish	Message Controller	Message
Ti									X	X			X			
Ta		X		X			X						X			
TL		X					X						X			
In			X						X		X	X		X		
DD			X								X		X			
D			X								X		X	X		
Me			X						X		X			X		
Mes								X		X					X	X
E		X					X	X	X				X			
WML																
WMS									X						X	
WI									X							
HML																
HMS							X								X	
HI																
CML																
CMS											X				X	
CI										X						
MML								X								
MMS								X		X					X	
MI								X								
IU											X	X				

The traceability matrix above illustrates the relationship between the various domain concepts and the software classes used in our application. As we know which domain concepts are linked to specific software classes, we can understand how they are related.

Design Patterns

Please refer to the Interaction Diagrams for design patterns per use case. This section explains the overall Design Pattern and how it improved the design for the application.

In all the Use Cases there is one Design Pattern used — Command Pattern. As the Command Pattern describes, the use cases separate the parameter preparation from program control (the decision of when to call).

Command Pattern is described to communicate through 3 identifiers: Client A → Command → Receiver B. Client A creates parameters for the command (parameter preparation) and Command executes them with Receiver B (program control). This method holds the advantage of decoupling parameter preparation from execution, amongst others.

Since all the Use Cases and a majority of all other communication between systems are executed through the Message Controller, the inference can be made that Command Pattern is used throughout the implementation of this application. The Message Controller represents the "Command" and the remaining two identifiers (Client A and Receiver B) can vary depending on the needed action.

Using Command Pattern as the primary Design Pattern improved the overall design due to the decoupling advantages from this design pattern—an advantage extremely important to us, especially since the application is developed by three different mini-groups. Separating the parameter preparation from the program control allows for less dependency between interfaces and equal dependency on the command. This reduces the possibility of chain error — errors occurring due to a chain reaction of another error from a different location.

This pattern also allows the command (the message controller) to have a uniform method signature which was convenient for the format of this design because changing the client would not require changing any receivers to accommodate because they communicate through the command. This method allows for us to further develop the client, command, or receiver without issues.

OCL Contracts

DatabaseAController

Invariants	❖ Context DatabaseAController inv: self.getPortNumber>=0

	● The port number that this database will listen on a non negative integer. ❖ Context DatabaseAController inv: self.getCurrentID>=0 ● The ID given to any employee must be a non negative integer.
Preconditions	❖ Context DatabaseAController::addEmployee(name : String, position : char) : Boolean pre: self.getEmployeeList() != null ● The employee list must be initialized before executing addEmployee. ❖ Context DatabaseAController::run() :Void pre: self.getCurrListener()!=null ● The current listener socket must be initialized before executing run. ❖ Context DatabaseAController::sendWaiters() :String pre: self.getEmployeeList() != null ● The employee list must be initialized before executing sendWaiters. ❖ Context DatabaseAController::loadEmployees() :void pre: self.getEmpFile() != null ● The employee file must be initialized with the path of the employee list text file before executing loadEmployees.

Post conditions	❖ Context DatabaseAController::addEmployee(name : String, position : char) :
	boolean post: let allValidPositions : Set = {'w','h','c','m','o'}
	if allValidPositions .exists(pos | position = pos) then
	self.getEmployeeList().size() = self.getEmployeeList().size()@pre+1
	else
	self.getEmployeeList().size() = self.getEmployeeList().size() @pre
	● If the position of the new employee is valid (is a waiter, host, chef, manager, or owner) then we will add it to the list and the employee list will grow by one. If the position is not then the employee list will remain the same size it was.

	❖ Context DatabaseAController::loadEmployees() :void post: self.getEmpFile() != null implies self.getEmployeeList().size()>0 ● If the employee file was initialized when executing loadEmployees, the employee list will now have at least one employee in it.

DatabaseBController

Invariants	❖ Context DatabaseBController inv: self.getPortNumber>=0 ● The port number that this database will listen on a non negative integer.
Preconditions	❖ Context DatabaseBController::addTable(tabNum: int, maxOccupancy: int, type : char) : boolean pre: self.getListOfTables() != null ● The table list must be initialized before executing addTable. ❖ Context DatabaseBController::run() : void pre: self.getCurrListener()!=null ● The current listener socket must be initialized before executing run. ❖ Context DatabaseBController::loadTablesFromFile() : void pre: self.getTabNumFile() != null ● The table number file must be initialized with the path of the table list text file before executing loadTablesFromFile.

Post Conditions	❖ Context DatabaseBController::addTable(tabNum: int, maxOccupancy: int, type : char) : boolean post: let allNumbersInUse : Set = self.getListOfTables.getKeys()
	if allNumbersInUse .exists(tableNumber | tabNum= tableNumber) then
	self.getListOfTables().size() = self.getListOfTables().size()@pre
	else
	self.getListOfTables().size() = self.getListOfTables().size() @pre+1

	❖ If the table number is already in the list of tables, the amount of tables in the list will still be the same, if you are adding a new table number then the list will increase in size by one. ❖ Context DatabaseBController::loadTablesFromFile() :void post: self.getTabNumFile() != null implies self.getListOfTables().size()>0 ● If the table list file was initialized when executing loadTablesFromFile, the table list will now have at least one table in it.

DatabaseCController

Invariants	❖ Context DatabaseCController inv: self.getPortNumber>=0 ● The port number that this database will listen on a non negative integer.
Preconditions	❖ Context DatabaseCController::addDishtoMenu(type: String, dishname:Strig, price:double) : int pre: self.menu != null ● The menu must be initialized before executing addDishtoMenu. ❖ Context DatabaseCController::decrementDish() : void pre: self.getDishData()!=null && self.getInventory()!=null ● The dish data list and the inventory must be initialized before executing decrementDish. ❖ Context DatabaseCController::loadMenuFromFile() : void pre: self.getMenuFile() != null ● The menu file must be initialized with the path of the menu text file before executing loadMenuFromFile.

Post Conditions	❖ Context DatabaseCController::addDishtoMenu(type: String, dishname:String, price:double) : int post: let allDishNames : Set = self.menu.getKeys()
	if allDishNames .exists(dishName | dishname= dishName) then
	self.menu().size() = self.menu().size()@pre
	else

	self.menu().size() = self.menu().size() @pre+1 ● If the dish is already in the menu, the amount of dishes in the menu will still be the same, if you are adding a new dish then the size will increase in size by one. ❖ Context DatabaseCController::loadMenuFromFile() :void post: self.getMenuFile() != null implies self.Menu.size()>0 ● If the menu file was initialized when executing loadMenuFromFile, the menu will now have at least one dish type in it.

ChefInterface

Invariants	❖ Context ChefInterface inv: self.getMCPortNumber>=0 ● The port number that this interface will listen on a non negative integer. ❖ Context ChefInterface inv: self.getEmpID>=0 ● The employee ID will be a non negative integer. ❖ Context ChefInterface inv: self.getCurrTicketNumber>=0 ● The current ticket number will be a non negative integer.
Preconditions	❖ Context ChefInterface ::chefTicketListener(ticket: Ticket) : void pre: self.ticketQueueUnstarted != null && self.getTicketLookUp()!= null ● The ticket Queue for unstarted tickets and the ticket look up list must be initialized before executing chefTicketListener. ❖ Context ChefInterface ::changeTicketLocation(oldStatus : char, t: Ticket) : void pre: let allValidStatus : Set = {'u','s','S','f'} allValidPositions .exists(stat | oldStatus= stat) ● The new status must be a valid status which stands for started, unstarted, semistarted or finished. ❖ Context ChefInterface ::setupMessageController() void pre: self.sender !=null ● The sender socket must be initialized before executing setupMessageController. ❖ Context ChefInterface ::logOut() void pre: self.sender !=null

	● The sender socket must be initialized before executing logOut. ❖ Context ChefInterface ::notifyManager() void pre: self.sender !=null ● The sender socket must be initialized before executing notifyManager.

Post Conditions	❖ Context ChefInterface ::chefTicketListener(ticket: Ticket) : void post:
	if ticket!=null then
	self.ticketQueueUnstarted.size() = self.ticketQueueUnstarted.size()@pre+1
	&& self.getTicketLookup().size() = self.getTicketLookup().size()@pre+1
	&& self.getCurrTicketNumber = self.getCurrTicketNumber@pre +1
	else
	self.ticketQueueUnstarted.size() = self.ticketQueueUnstarted.size()@pre
	&& self.getTicketLookup().size() = self.getTicketLookup().size()@pre
	&& self.getCurrTicketNumber = self.getCurrTicketNumber@pre
	● If the dish is already in the menu, the amount of dishes in the menu will still be the same, if you are adding a new dish then the size will increase in size by one.

<u>WaiterInteface</u>

Invariants	❖ Context WaiterInterface inv: self.getMCPortNumber>=0 ● The port number that this interface will listen on a non negative integer. ❖ Context WaiterInterface inv: self.getEmpID>=0 ● The employee ID will be a non negative integer.
Preconditions	❖ Context WaiterInterface ::logOut() void pre: self.sender !=null

	● The sender socket must be initialized before executing logOut. ❖ Context WaiterInterface ::setupMessageController() void pre: self.sender !=null ● The sender socket must be initialized before executing setupMessageController. ❖ Context WaiterInterface ::addGCorCoupon(price: double) :void pre:self.currTicket!= null ● The current ticket must be initialized before executing addGCorCoupon. ❖ Context WaiterInterface ::addDishToTicket(dish: Dish) :boolean pre:self.currTicket!= null ● The current ticket must be initialized before executing addDishToTicket. ❖ Context WaiterInterface ::notifyManager() void pre: self.sender !=null ● The sender socket must be initialized before executing notifyManager. ❖ Context WaiterInterface ::sendTicket(t: Ticket) void pre: self.sender !=null ● The sender socket must be initialized before executing sendTicket. ❖ Context WaiterInterface ::notifyWaiter(tableNumber: int) void pre: self.sender !=null ● The sender socket must be initialized before executing notifyWaiter. ❖ Context WaiterInterface ::paid(tableNumber: int) void pre: self.sender !=null ● The sender socket must be initialized before executing paid.
Post Conditions	❖ Context WaiterInterface ::loadMenu() :boolean post: self.jsonConverter() != null implies self.getMenu.size()>0 ● If the json converter was initialized when executing loadMenu, the menu will be able to be loaded and the menu attribute will no longer be empty. ❖ Context WaiterInterface ::addGCorCoupon(price: double) :void post:self.currTicket.price=self.currTicket.price @ pre+price ● The price of the current ticket will be changed by the price that is inputted into this function.

	❖ Context WaiterInterface ::addDishToTicket(dish: Dish) :boolean post:self.currTicket.listOfDishes.size()=self.currTicket.listOfDishes.size() @ pre+1 && self.currTicket.price=self.currTicket.price @pre +dish.price ● The amount of dishes is incremented by 1 and the price is also increased after this function is called. ❖ Context WaiterInterface ::removeDishFromTicket(dish: Dish) :boolean post:self.currTicket.listOfDishes.size()=self.currTicket.listOfDishes.size() @ pre-1 && self.currTicket.price=self.currTicket.price @pre -dish.price ● The amount of dishes is decremented by 1 and the price is also decreased after this function is called. ❖ Context WaiterInterface ::paid(tableNumber: int) :void post:self.getListOfTickets.size()=self.getListOfTickets.size() @ pre-1 ● The amount of tickets is decremented by 1 after this function is called.

ManagerInterface

Invariants	❖ Context ManagerInterface inv: self.getMCPortNumber>=0 ● The port number that this interface will listen on a non negative integer. ❖ Context ManagerInterface inv: self.getEmpID>=0 ● The employee ID will be a non negative integer.
Preconditions	❖ Context ManagerInterface ::logOut() void pre: self.sender !=null ● The sender socket must be initialized before executing logOut. ❖ Context ManagerInterface ::setupMessageController() void pre: self.sender !=null ● The sender socket must be initialized before executing setupMessageController. ❖ Context ManagerInterface ::sendMassNotification(content: String) void pre: self.sender !=null ● The sender socket must be initialized before executing sendMassNotification.

	❖ Context ManagerInterface ::addMessageToList(m: Message) void pre: self.listOfMessages !=null ● The list of messages must be initialized before executing addMessageToList. ❖ Context ManagerInterface ::deleteMessage(index: int) void pre: self.listOfMessages !=null ● The list of messages must be initialized before executing addMessageToList.

Post Conditions	❖ Context ManagerInterface ::deleteMessage(index: int) void post:
	If index<self.listOfMessages.size() then
	self.listOfMessages.size()=self.listOfMessages.size() @pre -1
	● The amount of messages in the list of messages will decrease by 1 as long as the index is valid
	❖ Context ManagerInterface ::addMessageToList(m: Message) void post:
	If m!=null then
	self.listOfMessages.size()=self.listOfMessages.size() @pre +1
	● The amount of messages in the list of messages will increase by 1 as long as the message is not null.

HostInterface

Invariants	❖ Context HostInterface inv: self.getMCPortNumber>=0 ● The port number that this interface will listen on a non negative integer. ❖ Context HostInterface inv: self.getEmpID>=0 ● The employee ID will be a non negative integer.
Preconditions	❖ Context HostInterface ::logOut() void pre: self.sender !=null ● The sender socket must be initialized before executing logOut. ❖ Context HostInterface ::setUpMessageController() void pre: self.sender !=null ● The sender socket must be initialized before executing setUpMessageController.

	❖ Context HostInterface ::notifyManager() void pre: self.sender !=null ● The sender socket must be initialized before executing notifyManager. ❖ Context HostInterface ::notifyWaiter(m: Message) void pre: self.sender !=null ● The sender socket must be initialized before executing notifyWaiter. ❖ Context HostInterface ::sendSeated(waiterId: long,tableNumber: int) void pre: self.sender !=null ● The sender socket must be initialized before executing sendSeated.

Post Conditions	❖ Context HostInterface ::loadTables() :boolean post: self.jsonConverter() != null implies self.allTables.size()>0
	● If the json converter was initialized when executing loadTables, the allTables will be able to be loaded and the table list attribute will no longer be empty.
	❖ Context HostInterface ::seat(waiterName: String, tableNumber: int) :boolean post:
	let t= self.allTables.get(tableNumber)
	if t.status== 'r' then
	t.status='s' &&
	self.readyTables.size()=self.readyTables.size() @pre -1 &&
	self.seatedTables.size()=self.seatedTables.size() @pre +1
	else
	self.readyTables.size()=self.readyTables.size() @pre &&
	self.seatedTables.size()=self.seatedTables.size() @pre
	● When this function is called, it checks if the table selected is in the ready state and if it is, the list of ready tables decreases by one and list of seated tables increases by 1 otherwise the sizes stay the same.
	❖ Context HostInterface ::paid(tableNumber: int) :boolean post:
	let t= self.allTables.get(tableNumber)
	if t.status== 's' then
	t.status='p' &&
	self.seatedTables.size()=self.seatedTables.size() @pre -1 &&
	self.paidTables.size()=self.paidTables.size() @pre +1

	else
	self.seatedTables.size()=self.seatedTables.size() @pre && self.paidTables.size()=self.paidTables.size() @pre
	● When this function is called, it checks if the table selected is in the seated state and if it is, the list of seated tables decreases by one and list of paid tables increases by 1 otherwise the sizes stay the same.
	❖ Context HostInterface ::cleaned(tableNumber: int) :boolean post:
	let t= self.allTables.get(tableNumber)
	if t.status== 'p' then
	t.status='r' &&
	self.paidTables.size()=self.paidTables.size() @pre -1 &&
	self.readyTables.size()=self.readyTables.size() @pre +1
	else
	self.paidTables.size()=self.paidTables.size() @pre && self.readyTables.size()=self.readyTables.size() @pre
	● When this function is called, it checks if the table selected is in the paid state and if it is, the list of paid tables decreases by one and list of ready tables increases by 1 otherwise the sizes stay the same.

MessageController

Invariants	❖ Context MessageController inv: self.getPortNumber>=0 ● The port number that this interface will listen on a non negative integer.
Preconditions	❖ Context MessageController ::addWaiterSender(id: long, sender: MessageControllerSender) void pre: self.waiterOut !=null ● The waiterOut socket must be initialized before executing addWaiterSender. ❖ Context MessageController ::addHostSender(id: long, sender: MessageControllerSender) void pre: self.hostOut !=null ● The hostOut socket must be initialized before executing addHostSender. ❖ Context MessageController ::addChefSender(id: long, sender: MessageControllerSender) void pre: self.chefOut !=null

	● The chefOut socket must be initialized before executing addChefSender. ❖ Context MessageController ::addManagerSender(id: long, sender: MessageControllerSender) void pre: self.managerOut !=null ● The managerOut socket must be initialized before executing addManagerSender. ❖ Context MessageController ::removeWaiterSocket(id: long, sender: MessageControllerSender) void pre: self.waiterOut !=null ● The waiterOut socket must be initialized before executing removeWaiterSocket. ❖ Context MessageController ::removeHostSocket(id: long, sender: MessageControllerSender) void pre: self.hostOut !=null ● The hostOut socket must be initialized before executing removeHostSocket. ❖ Context MessageController ::removeChefSocket(id: long, sender: MessageControllerSender) void pre: self.chefOut !=null ● The chefOut socket must be initialized before executing removeChefSocket. ❖ Context MessageController ::removeManagerSocket(id: long, sender: MessageControllerSender) void pre: self.managerOut !=null ● The managerOut socket must be initialized before executing removeManagerSocket.
Post Conditions	❖ Context MessageController ::addWaiterSender(id: long, sender: MessageControllerSender) void post: self.waiterOut.size()=self.waiterOut.size() @pre +1 ● The list of waiter output streams will increase by 1. ❖ Context MessageController ::addHostSender(id: long, sender: MessageControllerSender) void post: self.hostOut.size()=self.hostOut.size() @pre +1 ● The list of host output streams will increase by 1. ❖ Context MessageController ::addChefSender(id: long, sender: MessageControllerSender) void post: self.ChefOut.size()=self.ChefOut.size() @pre +1

	● The list of chef output streams will increase by 1. ❖ Context MessageController ::addManagerSender(id: long, sender: MessageControllerSender) void post: self.managerOut.size()=self.managerOut.size() @pre +1 ● The list of manager output streams will increase by 1. ❖ Context MessageController ::removeWaiterSocket(id: long, sender: MessageControllerSender) void post: self.waiterOut.size()=self.waiterOut.size() @pre -1 ● The list of waiter output streams will decrease by 1. ❖ Context MessageController ::removeHostSocket(id: long, sender: MessageControllerSender) void post: self.hostOut.size()=self.hostOut.size() @pre -1 ● The list of host output streams will decrease by 1. ❖ Context MessageController ::removeChefSocket(id: long, sender: MessageControllerSender) void post: self.ChefOut.size()=self.ChefOut.size() @pre -1 ● The list of chef output streams will decrease by 1. ❖ Context MessageController ::removeManagerSocket(id: long, sender: MessageControllerSender) void post: self.managerOut.size()=self.managerOut.size() @pre -1 ● The list of chef output streams will decrease by 1.

Data Types & Operation Signatures

Changes we made in Data Types & Operation Signatures

- Since we were not able to implement SQL at this time, we used Java classes to hold data in the databases. The classes are labeled as databaseA,B,C controllers.
- Instead of having functions that would listen for messages from the message controller and send messages to the message controller in each employee interface, we had two separate classes for each employee interface (that were the <EmployeePosition>MessageListener and <EmployeePosition>MessageSender) that deals with communication with the message controller .
- Added JFrame to display the main screens for each employee

- For the chef interface, tickets were separated (based on status) into four ticket queues (unstarted, semi-started, started, finished) , instead of one queue. A hashmap that maps the ticket number to the ticket was also added.
- Added function to load menu items from the database for the waiter
- Add a notify manager function to each employee interface
- Added functions to modify the ticket for the waiter interface
- For the host interface, tables were separated (based on status) into 3 lists (ready, paid, seated), instead of two lists. It also has a list of waiters. It has a loadTables function to load the list of tables from the database for the host.
- For the manager interface, add the mass notification function to send a notification to all the employees.
- For all the employee interfaces, add the addNotification function to add the notification to the current screen.
- Used the separate classes to display the panels (the graphical user interfaces) for the employees using JPanel

Waiter Interface

Attributes:

+sender: WaiterMessageSender	The sender is of object WaiterMessageSender type which sends messages to the message controller to be forwarded to the correct employee.
+DBCSender: DataBaseCSender	The DBCSender is of object DataBaseCSender type which sends ticket auditing messages to database C to be recorded.
+frame: JFrame	Used to display the screen of the waiter's tablet
+empID: long	The employee id used to help the Message Controller identify the tablet the waiter is using.
+name: String	The name of the employee displayed on the screen
+loggedOut: boolean	The status that represents the log in status of the employee, When this is true the screen returns from the waiter page back to the log in page. When this is false, it stays on the waiter's page.
+currTicket: Ticket	The current ticket opened, null if none is opened
+ menu: Menu	The menu is of object Menu type which contains the list of items in the menu.

+ listOfTickets: HashMap<Integer,Ticket>	HashMap that holds the list of tickets this waiter is in charge of. The key is an Integer which represents the table number the ticket is from. The value is the ticket.
+ticketListScreen: WaiterTickListScreen	The tickListScreen is of object WaiterTickListScreen that is a panel that draws the screen for the waiter's list of tickets.
+oneTickScreen: WaiterOneTicketScreen	The oneTickScreen is of object WaiterOneTicketScreen that is a panel that will be used when one ticket is selected from the list of tickets the waiter is in charge of. Displays the ticket as well as the menu. This is where the waiter will place the order.

Functions:

+void openTicketScreens(int tableNumber)	Opens the ticket for the corresponding table number. Switches from the list of tickets screen to one open ticket screen for the waiter. The tableNumber is the table number of the ticket you wish to open.
+ void paid(int tableNumber)	This function is called when a table has paid and the waiter clicks the paid button. This would alert the host and take the ticket with that table number off the waiter's list.The tableNumber is the table number of the table that just paid.
+ boolean loadMenu()	Loads the menu from database C for the waiter.
+void logOut()	Logs the waiter out. Sends a message to the Host and the Message Controller to notify that waiter is logging out.
+void notifyManager(Ticket currTicket2)	Sends a notification to the Manager that the table for that ticket needs help
+void sendTicket(Ticket t)	Sends the ticket to the Chef and marks the sent items

+void addGCorCoupon(double price)	Adds a GiftCard or a Coupon to the ticket and the price of the gift card or the amount to deduct from the price of the meal using the coupon.
+boolean addDishToTicket(Dish dish)	Adds the given dish to the ticket that is currently selected.
+void removeDishFromTicket(int indexInTicket)	Removes the dish at the given index on the ticket that is currently selected.
+void addComment(int ind, String com)	Adds comment com to the dish at index ind on the curr ticket
+void backToMainScreen()	Switches the screen from an open ticket screen to the list of tickets screen, the main page for the waiter.
+void addNotification(String content)	Adds a notification on the current screen by calling another method in panel.
+void updateScreen()	Updates the current panel for the waiter and makes it redraw all the buttons.
+void setUpMessageController()	Sets up the Message Controller which controls the communication between the employees and alerts it that waiter is logged in.
+void notifyWaiter(int tableNumber)	The host interface would use this function to notify the waiter that his table needs his service.
+void removeLowInventoryDishes(String[] dishes)	This is called when the threshold for low inventory is met and the chef can no longer make these dishes. This function removes the array of dish names from the menu.

Chef Interface

Attributes:

+empID: long	The employee id used to help the Message Controller identify the tablet the employee is using.
+name: String	The name of the employee displayed on the screen
+loggedOut: boolean	The status that represents the log in status of the employee, When this is true the screen returns from the waiter page back to the log in page. When this is false, it stays on the waiter's page.
+currTicketNumber: long	The current ticket number of the chef
+ ticketLookup: HashMap<Long, Ticket>	Hashmap that maps the ticket number to the ticket
+ticketQueueUnstarted: ArrayList<Long>	The ticket queue of tickets in which all the dishes of the ticket have not been started.
+ticketQueuesemiStarted: ArrayList<Long>	The ticket queue of tickets in which some of the dishes of the ticket have been started
+ticketQueueStarted: ArrayList<Long>	The ticket queue of tickets in which all the dishes of the ticket have been started.
+ticketQueueFinished: ArrayList<Long>	The ticket queue of tickets in which all the dishes of the ticket are finished.
+loginPanel: LogInScreen	The loginPanel is of LogInScreen type which is used to display the login screen for the chef.

Functions:

+ void chefTicketListener(Ticket ticket)	Function used to read the incoming tickets from the waiters and places them in the queue that the chef can view.
+ void decrementInventoryForDish(Dish dish)	Decrements the amount of each item in the inventory by the proper amount of ingredients for this dish.

+ void changeTicketLocation(char oldstatus, Ticket t)	Changes the location of the ticket based on the status of it.It takes the ticket off of one list and adds it to another.
+ void setUpMessageController()	Sets up the Message Controller which controls the communication between the employees and alerts it that chef is logged in.
+void openTicketScreens(long ticketNumber)	Opens the ticket on the chef screen and allows him to view the contents of it and edit the status of each dish on it.
+ void backtoMainScreen()	Switches the screen from an open ticket screen to the list of tickets screen.
+ void logOut()	Sends a message to the Host and MC to notify that chef is logging out.
+void notifyManager()	Sends a notification to the manager
+void updateScreen()	Updates the screen to the main chef screen

Host Interface

Attributes:

+allTables: HashMap<Integer,Table>	Hashmap that maps the table number to the table. It holds all the tables in the restaurant.
+ readyTables: ArrayList< Integer>	List of all the ready tables in the order they will be displayed on the host's screen. The Integer is the table number.
+seatedTables: ArrayList<Integer>	List of all the seated tables in the order they will be displayed on the host's screen. The Integer is the table number.
+paidTables: ArrayList<Integer>	List of all the tables that have paid the check and need to be cleaned. The Integer is the table number.

+listOfWaiters: HashMap<String, Integer>	Hashmap that maps the name of the waiter to the ID of the waiter.
+waiterTotalTables: HashMap<String, Integer>	Hashmap that maps the name of the waiter to the total number of tables he currently has.
+waiterOfTable: HashMap<Integer,String>	HashMap that maps the table number to the waiter who has it
+loginPanel: LogInScreen	The loginPanel is of LogInScreen type which is used to display the login screen for the host.
+empID: long	The employee id used to help the Message Controller identify the tablet the host is using.
+name: String	The name of the employee displayed on the screen
+sender: HostMessageSender	The sender is of object HostMessageSender type which sends messages to the message controller to be forwarded to the correct employee.
+tableScreen: HostTableScreen	Displays the screen of the host's tablet which would be the tables of the restaurant

Functions:

+boolean loadTables()	Loads the list of tables from Database B.
+void setUpMessageController()	Sets up the Message Controller which controls the communication between the employees and alerts it that host is logged in.
+void notifyManager()	Sends a notification to the manager
+void notifyWaiter(Message m)	Send a notification to the waiter
+int seat(String waiterName, int tableNumber)	Seats the table with the corresponding table number with the waiter whose name is one of the parameters.

+void sendSeated(long waiterId, int tableNumber)	Sends a message to the waiter with the corresponding waiter id that his table (the table number is one of the parameters) just got sat.
+void addNotification(String content)	Adds a notification on the host's screen
+void logOut()	Logs the host out. Sends a message to the Message Controller to notify that the host is logging out.
+void updateScreen()	Updates the current panel - makes them redraw all the buttons
+void paid(int tableNumber)	Moves the seated table to the paid section when the table pays the check, gives the host an idea of what tables to clean for the next set of customers
+void cleaned(int tableNumber)	Moves a table that was just cleaned from the paid section into the ready list
+boolean loadWaiters()	Loads waiters and IDs from database A.
+long getWaiterIdForTable(int tableNumber)	To find the ID of the waiter who serves the table with the corresponding table number given

Manager Interface

Attributes:

+listOfMessages: LinkedList<Message>	List of messages that the Manager has received in chronological order. End of the list is most recent
+sender:ManagerMessageSender	The sender is of object ManagerMessageSender type which sends messages to the message controller to be forwarded to the correct employee.
+frame: JFrame	Used to display the screen for the manager
+empID: long	The employee id used to help the Message Controller identify the tablet the manager is using.
+name: String	The name of the employee displayed on the screen

+loggedOut: boolean	The status that represents the log in status of the employee, When this is true the screen returns from the manager's page back to the login page. When this is false, it stays on the manager's page.
+loginPanel:LogInScreen	The loginPanel is of LogInScreen type which is used to display the login screen for the manager.
+manScreen:ManagerScreen	The manScreen is of ManagerScreen type and is used to display the screen for the manager once he is logged in as manager.

Functions:

+ void logOut()	Logs the manager out. Sends a message to the Message Controller to notify that the manager is logging out
+void updateScreen()	Updates the current panel - makes them redraw all the buttons
+void setUpMessageController()	Sets up the Message Controller which controls the communication between the employees and alerts it that the manager is logged in.
+void deleteMessage(int index)	Deletes the message at the given index in the message list. Caller should be sure to check if the index is valid.
+void sendMassNotification(String content)	Sends a mass notification to all waiters, hosts, and chefs
+void addMessageToList(Message m)	Adds this message to the list of messages that the manager can view

Message Controller

Attributes:

+waiterOut HashMap<Long,MessageControllerSender>	List of output streams(senders) of the waiters logged in with the employee id as the key
+hostOut HashMap<Long,MessageControllerSender>	List of output streams(senders) of hosts logged in with employee id as key
+ chefOut HashMap<Long,MessageControllerSender>	List of output streams(senders) of chefs logged in with employee id as key
+managerOut HashMap<Long,MessageControllerSender>	List of output streams(senders) of managers logged in with employee id as key

Functions:

+ void addWaiterSender(long id,MessageControllerSender sender)	Adds a waiter to the list of output streams
+ void addChefSender(long id,MessageControllerSender sender)	Adds a chef to the list of output streams
+ void addHostSender(long id,MessageControllerSender sender)	Adds a host to the list of output streams
+ void addManagerSender(long id,MessageControllerSender sender)	Adds a manager to the list of output streams
+void removeWaiterSocket(long id)	Removes a waiter from the list of output streams
+void removeChefSocket(long id)	Removes a chef from the list of output streams
+void removeHostSocket(long id)	Removes a host from the list of output streams
+void removeManagerSocket(long id)	Removes a manager from the list of output streams

Database A Controller

Attributes:

+currListener Socket	Socket to one tablet that database A will use to log employees on.
+currentID long	Holds the next ID number for a newly created employee.This ensures that each employee ID is unique.
+employeeList ArrayList<Employee>	Array of Employees which matches the Employee ID (int) with the employee

Functions

+boolean addEmployee(String name, char position)	Adds an employee with his position to the database
+String sendWaiters()	Generates a string of all the waiters logged into the database. Uses the waiter's name and ID
+void loadEmployees()	Loads the employee list from a file

Database B Controller

Attributes:

+currListener Socket	Socket to one tablet that the database B controller would listen to
+listOfTables TableList	Holds the list of tables in the restaurant

Functions:

+boolean addTable(int tabNum, int maxOccupancy, char type)	Adds table to the list of tables of the restaurant
+void loadTablesFromFile()	Loads the tables from the file

Database C Controller

Attributes:

+inventory HashMap<String, Ingredient>	Models the inventory of the restaurant. Maps the name of the ingredient to the ingredient.
+dishData HashMap<String, DishData>	Maps the name of the dish to its dish data which is information about the dish. This will be used to figure out the amount of each ingredient to decrement from the inventory for a started dish.
+waiterSenders HashMap<Integer,Sender>	List of all the waiter senders that are connected to database C.
+chefSender Sender	Chef sender that is connected to database C
+menu Menu	Holds the menu. The waiter will need this when logging on.

Functions:

+boolean addIngredientToInventory(String ingredientName,Double amountLeft, String unitOfAmount, Double threshold)	Adds the ingredient to the inventory along with information about the ingredient such as the amount of ingredient you have at the time, the unit the ingredient is measured in, and the threshold, amount that determines low inventory.
+int addDishtoMenu(String type, String dishname, double price)	Adds a dish to the menu.
+ void recordTicket(String tick)	Adds the ticket to the records
+void decrementDish(String dishName)	Decrements all ingredients in the dish by the proper amount in the inventory and if the threshold is met, it will generate the proper notifications.
+void sendLowInventoryNotifications(Ingredient i)	Sends the chef a low inventory notification. Sends the waiter interfaces a message that it will take off the dishes with ingredient i in their menu.
+void loadMenuFromFile()	Loads the menu from the menu file.

System Architecture & System Design

Changes we made in System Architecture and System Design:

- Subsystems are changed: Database A holds the employee list, Database B holds table queue and Database C holds inventory and menu. Database A will be accessed by all employees, B will be accessed by the host and C will be accessed by chef and waiter.
- For saving employee information and ticket information for future use, we used textfiles with Java rather than SQL. We plan to implement SQL in the future because as of now the amount of information we are storing is limited as we are still in the testing stages of the application.

Architectural Styles

The design we use is a client server model with distributed databases. Different clients connect with multiple integrated servers and we have multiple databases that hold different kinds of data depending on user of that database. We used distributed databases to reduce the overhead for each individual database.

The databases are the servers which are used by their respective clients. We used three different databases. The first one holds employee information as well as other static information that can be loaded once a day. This is accessible only to the owner of the restaurant as it contains administrative information. The other two databases are accessed by employees and, they hold ticket orders, dish information, list of tables, inventory, etc.

We automated the restaurant experience by allowing each client to connect with the server and have access to the information needed for their job function. Each employee login allows them to gain access to required information for their job. For example, when the host logs in, he has access to the list of tables and the waiter associated with each table but not any dish information since that does not pertain to his job. When a waiter logs in, he only has access to table ticket information. Using the distributed databases, we can limit the information accessible to a specific employee based on their job function thus limiting data corruption. Since we are not sharing any information between databases, synchronization is not required.

In addition to utilizing databases, we also implement a messaging system for client devices to communicate with each other for their respective functions. Rather than having the extra overhead of sending messages, we allow the client devices to send messages to the message controller and it handles the sending and receiving of messages, as explained by the Expert Doer Principle. With this system, we are enhancing communication between users for simplicity and ease of use.

Since each object has minimal interactions with other objects, if one of them has an issue, then it will not impair the efficiency of the other users. For example, if there is a problem with the interface associated with the host, the functionality of other objects such as waiter and chef should not be affected. This helps make maintenance of the system an easier task.

Overall, we chose to use the client server model with distributed databases because it reduces overhead for each subsystem, multiple clients are able to connect with the server simultaneously and makes maintenance easier.

Identifying Subsystems

Observe Figure A below.

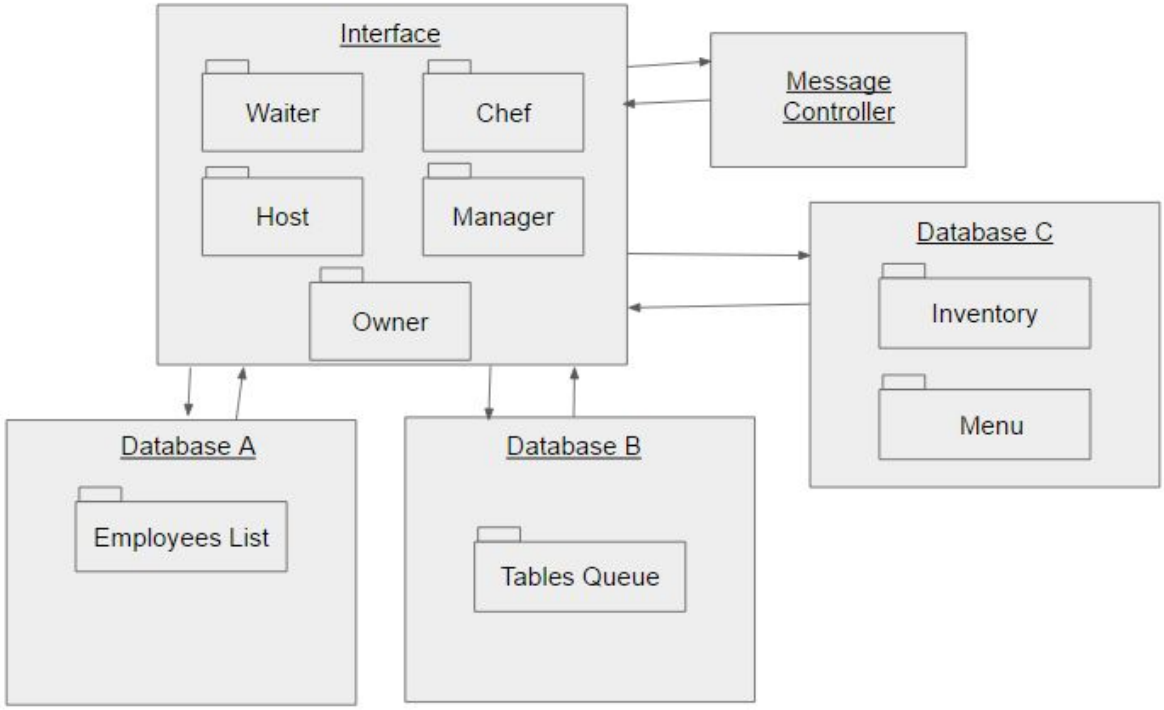


Figure A: Package Diagram

Our package diagram consists of all elements of the subsystems. The interface package contains all employee interfaces and gives each employee different options and accessibility depending on his specific job function. We use a distributed system so we have three separate databases. Database A holds static information that is not changed on a daily basis. This holds the list of employees. Database B, which is used primarily by the Host interface, holds all tables and their statuses. Database C, which is primarily used by the Chef, holds the inventory and menu. The message controller, accessed by all employees, through which they send and receive messages and alerts.

Mapping Subsystems to Hardware

We have chosen to use the client-server model. The server runs on a master computer and our application runs on each employee's tablet. The servers are all systems that clients connect to such as databases and message controller. The application contains a message controller and interface processing. This allows users to access and modify data that is accessible to them depending on their job function.

Persistent Data Storage

Our system requires data to be stored and accessible longer than the amount of time that the application runs. Once data such as employee information, menu items and tables numbers have initially been stored in the databases, we require it to be available for future use and documentation. Currently we use text files with Java to store data, in the future we plan to switch to the SQL database. The SQL database allows us to persistently store data in tables and saves that data to the local files on the computer. This ensures the long time reliability and maintenance of the data which is extremely important for our application. We save all employee information, menu items and ticket information to files on the local computer in order to have access to them in the future. All ticket receipt information of paid tickets is saved in text files. The storage of ticket information allows the owner to go back and look at statistics or for auditing.

Network Protocol

Our future plan is to implement Microsoft Azure SQL database. This will allow us to create a way to store information in a cloud based server. Unlike other cloud based databases, the SQL database allows us to make relational queries with the stored data such as searching, data analysis and data synchronization. For our application, *Serve with Ease*, this type of capability is extremely important because it allows us to store a virtually unlimited amount of data on the cloud as well as have an easy way to modify that data. As far as communication between processes go, we will be using TCP protocol. We will use Java's sockets.

Algorithms & Data Structures

Algorithms

Changes we made in Algorithms:

- Replaced Host event listener with host interface and java swing buttons. Now we use smaller functions for each type of table switch.

Generating Host's Table Queue:

The mathematical model and algorithm for how the host's list of tables is generated are very similar. The algorithm is event driven as events are what cause the changes in the lists. The host screen will show ready tables (tables that are open) in one list and not ready tables (tables that are seated or dirty/paid) in another. The second list of not ready tables should have the paid tables on top (as those are the most important to the host since he must clean them soon). In order to keep track of order easily we made three ArrayLists of integers, where each integer represented the table number. We used two arraylists to make up the not ready tables because this will make it much easier to move a table from seated to paid. All we would need to do is remove that table from the seated arraylist and add it to the end of the paid arraylist. Then when the host interface prints the screen, it will show the notready list as the oncheck list followed by the seated list. We used an arraylist of integers as opposed to tables because when you are switching a table from seated to paid and you need to change what list it is on, it is faster to transfer an integer as opposed to a whole table object. To make this model work we added an all tables hashmap. This serves as a lookup table for table numbers to tables and shortens access time.

Pseudo Code: This code is based off of code in HostInterface.java

```
//Hashmap links integer (table #) to its table object - holds all the tables in the restaurant
HashMap<Integer, Table> allTables;

//list of all the ready tables in order they will be displayed
ArrayList< Integer> readyTables;

//list of tables that are either seated
ArrayList< Integer> seatedTables;

//list of tables that need to be cleaned
ArrayList< Integer> paidTables;

//Constructor:
public HostInterface(){
  //Code about logging in….
  allTables = getMapofTablesFromDataBase();
  readyTables = new ArrayList<Integer>();
  seatedTables = new ArrayList<Integer>();
  paidTables = new ArrayList<Integer>();
  Iterator<Integer> it = allTables.keySet().iterator();
  while( it.hasNext()){
          Integer key= it.next();
          Table table = allTables.get(key);
```

```
            table.setStatus('r');
            readyTables.add(key);
    }
}
//This function is called when a waiter sends the host a notification that their table has paid
public void paid(int tableNumber) {
            Table t = allTables.get(tableNumber);
            t.status = 'p'; //set status to paid
            for(int i=0; i<seatedTables.size();i++){
                        int curr = seatedTables.get(i);
                        if(curr == tableNumber){
                                    seatedTables.remove(i);//remove the seated table from the seated list
                                    paidTables.add(tableNumber); //add it to the paid list
                                    break;
                        }
            }
            updateScreen(); //redraw the lists
    }

//This function is called when the host clicks the seat button
public int seat(String waiterName, int tableNumber){
            Table t = allTables.get(tableNumber);
            if (!(t.status=='r')){ //if table is not ready
                        return -1;
            }
            t.status = 's';//change status to seated
            for(int i =0; i<readyTables.size(); i++){
                        if(readyTables.get(i) == tableNumber){
                                    readyTables.remove(i);//take table off the ready list
                                    seatedTables.add(tableNumber);//add it to seated list
                                    break;
                        }
            }
            sendSeated(waiterName, tableNumber);
           //this will send a message to the waiter that they have been recently sat
            updateScreen();
            return 0;
    }

//This function is called when the host clicks the cleaned button
public void cleaned(int tableNumber) {
            Table t = allTables.get(tableNumber);
            t.status = 'r';
```

```
        for(int i=0; i<paidTables.size();i++){
                int curr = paidTables.get(i);
                if(curr == tableNumber){
                        paidTables.remove(i); //remove table from paid list
        readyTables.add(tableNumber) ;//add it to the paid list
        break;
                }
        }
        updateScreen(); //redraw the lists
}
```

Generating a Ticket:

Things we changed:

- Renamed generating the cost of the ticket to generating a ticket.
- We replaced the waiter event listener with the waiter interface and java swing buttons as well as functions to handle the the pushing of waiter buttons.
- We still use the menu but instead of mapping the name of the dish to it's dish data we map it to a dish object. This does not include the ingredients of the dish only the price. We made this switch because the waiter does not need to know the ingredients or the amount of ingredients that are needed for the dish when generating a ticket, only the price and the name.
- Dish data is still used but only by Database C as this is what keeps track of inventory.
- Also when removing a dish from the ticket we now use the index of the dish on the ticket instead of it's name, which makes removing must faster. We also implemented modifications to dishes, as well as coupons and gift cards to tickets.

This algorithm is used to generate the ticket that the waiter sees on his screen. Our algorithm is event driven since the ticket should change only when the waiter adds or removes items from a ticket. In the waiter interface we have a field called menu which is a hashmap that maps to hashmaps. The outer map maps the type of dish to a hashmap of all the dishes of that type. The key of the outer hashmap is the type of dish and the data is a hashmap of dishes of that type. The inner map maps the dish name to the dish object. The inner hashmap key is the name of the dish and the data associated with the key is the dish object. The dish object is what holds the price of the dish amongst other things. There is another field currDishType in the waiter interface which is used to keep track of what tab is open when the waiter is placing the order (ie appetizer, entree, dessert). This will allow the application to determine the type of dish that the waiter is adding to the ticket when he clicks the button with the dish name on it. This will make looking up the dish in the menu hashmap easier because we will already know what the type of dish that we are trying to find is. In the event that a waiter wants to remove a dish from the ticket, the

ticket will contain the index of that dish on the ticket we will be able to use that to remove that dish of the ticket.

Pseudo Code: This code is based off of code in WaiterInterface.java

```
//current ticket open, null if none is open
public Ticket currTicket;

//First String maps category (Drink, App, Entree, Dessert) to a map
//Inner map maps dish name to dish
public HashMap<String, HashMap<String,Dish>> menu;

//Name of waiter
public String name;

//Constructor
public WaiterInterface(){
        //set up messaging system and other credentials
        ...
        menu = loadMenuFromDataBaseC();
}
//This function adds the dish to the ticket. It is called when a waiter clicks on a dish from the on //his screen.
public boolean addDishToTicket(Dish dish) {
        //make a copy of the ticket so
        currTicket.addDishToTicket(dish.makeCopyOfDish()); //see Ticket.java below
        updateScreen();
         return true;
  }

//This function removes the dish at index indexInTicket from thet ticket. It is called when a waiter //clicks the remove button. Before calling this the applicaiton checks if the dish they are trying to //remove has already been sent to the chef (in which case it can not be deleted and so this //function will not be called).
public void removeDishFromTicket(int indexInTicket) {
         currTicket.removeDishFromTicket(indexInTicket);//see Ticket.java below
  }

//This will add a comment to the dish at index ind on the current ticket. It is called when a waiter //clicks on the modification button and selects a modification to add to a dish.
public void addComment(int ind, String com) {
         Dish d =currTicket.listOfDishes.get(ind);
         if(d.getStatus()!='c')//can't add a comment to a gc or coupon
                 d.comments.add(com);
  }
```

```
//This function will add a GiftCard or Coupon to the current ticket. A positive price will add a gift //card
of that amount and a negative will add a coupon of that amount. It will be called when the //waiter hits the
gift card or coupon button and enters the amount in.
public void addGCorCoupon(double price){
        if(price==0){
                Return; //don't need to do anything
        }
        else if(price>0){ //GIFT CARD
//generate a dish with the name Gift Card and the price equal to the parameter price. The third //argument
is a string to represent the type of dish use null because it is not an entree, drink, //etc)
                Dish gc = new Dish("Gift Card", price, null);
                gc.changeStatus('c');
//this marks this dish as a coupon/GC so it will not be on the ticket when it is sent to the chef
                currTicket.listOfDishes.add(gc);//add the Gift card dish to the ticket to display
                currTicket.price=currTicket.price + gc.price; //increase the ticket price
        }
        else{
                //COUPON
//generate a dish to represent the coupon (notice price here is negative)
                Dish comp = new Dish("Coupon", price, null);
                comp.changeStatus('c');//mark it as a coupon
                currTicket.listOfDishes.add(comp); //add it to the ticket
                currTicket.price=currTicket.price + comp.price;//decrease the price
        }
  }
```

Ticket.java

```
//Adds the dish to the ticket and increases its price
public void addDishToTicket(Dish d){
        listOfDishes.add(d);
        amountOfDishes++;
        this.price=this.price + d.price;
}

//Removes a dish from a ticket and decrease its price
public boolean removeDishFromTicket(int indexOfDishInTickList){
        if(indexOfDishInTickList<0 || indexOfDishInTickList>amountOfDishes){
                return false;
        }

        Dish del =listOfDishes.get(indexOfDishInTickList);
        if(del==null || del.sent){
```

```
                return false;
        }
        this.price=this.price - del.price;
        if(del.getStatus() =='c'){
                listOfDishes.remove(indexOfDishInTickList);
                return true;
        }
        amountOfDishes= amountOfDishes-1;
        char s = del.getStatus();
        listOfDishes.remove(indexOfDishInTickList);
        return true;
}
```

Updating Inventory

Things we changed:

- Now the chef has a direct link to database C so it can continually send and receive messages from database C without going through the message controller. This means that the chef interface will send the name of the dish the chef just started to the database C. Then, Database C will update the inventory and if the threshold is met then it will send out a message to the chef. Then the chef will forward it to the manger. Other than that the algorithm is the same.

Updating the inventory is an event driven algorithm. This is because the inventory only changes when the chef starts cooking the dish. This is signified when the chef changes the status of a dish from unstarted to started. This event is caught by the chef event listener (shown above) and it calls the function decrementInventoryForDish to decrease the stock of each ingredient for that dish. Notice that the amount of each ingredient that is needed to be decreased from the inventory is stored in dish data. This is how we know the appropriate amount of ingredient to decrease by.

Pseudo Code:

```
//Maps Ingredient Name to Ingredient
private HashMap<String, Ingredient> inventory;

private void decrementInventoryForDish(Dish dish) {
        DishData d= dish.getDishData();
        String[] ingredients= d.getListOfIngredients();
        for(int i=0; i<ingredients.length; i++){
                //decrement each ingredient of this dish in the inventory
                inventory.get(ingredients[i]).decrementAmountBy(d.getAmount(ingredients[i]));
        }
```

```
  }
```

Notice this is called by the chef event listener (listed above in Generating Chef's Ticket Queue).

```
if(e.newStatusOfDish == 's'){
        decrementInventoryForDish(e.dish);
}
```

Data Structures

Our system uses many complex data structures such as arraylists and hashmaps. We choose to use these structures due to their flexibility and mass data storage traits. Hashmaps allow us to categorize and store data in an easily accessible manner. It is flexible and malleable, allowing easy inputs of new data. Arraylists allow us to maintain track of information in a structured manner. Since they automatically increase in size when needed, there is little risk of depleting the space available. We also implement many queues for keeping the list of tables and list of tickets.

User Interface Design & Implementation

Things We Changed:

- Changes are described under each section, please read below.

1. **Waiter Screen**

Observe Figures Wa and Wb below. Figure Wa represents the waiter's home screen, designed for easy use. Figure Wb represents the waiter's screen once they click on a table. These screens are used in 2 of the 3 Use Cases described: Closing a Table and Placing an Order.

From our initial mockup we have improved the Wa screen by adding the Notify Waiter option which allows a waiter to notify another waiter that their table needs them.

From our initial mockup we have improved the Wb screen by adding more options, such as Modify, which holds numerous options for the waiter to add to a dish. The Notify Manager option was changed to just Notify and allows the waiter to select more specific categories for issues within it. Along with the Print button to print the receipt, we have also have a the Paid button. The host will be updated when the waiter clicks this. We have removed the Back button for the Remove button. We have also included 2 buttons to handle Coupons and Gift cards.

Figure Wa: Waiter Home Screen

Figure Wb: Waiter Screen When Placing an Order

2. Host Screen

Observe Figure H below. This represents the host screen, designed with the goal of having low complexity during use. This screen is not used directly in the Use Cases mentioned but it is affected from actions committed in Use Case 1.

From our initial mockup we have improved this screen by adding more options to the bottom of the screen, such as Seat and Cleaned. In the initial mockup with planned to write the status of the table next to the number but we have improved on that now. We have color coded the tables on the status of the table's progress in their dining experience (green for ready to be seated, red for seated, and yellow for paid/need to be cleaned). This allows the host to visually understand all the tables without having to scan and read for the status. We have also added a list of actively working waiters to the bottom right of the screen and a count of how many tables they are handling.

Figure H: Host Screen

3. Manager Screen

Observe Figure Ma and Mb below. Ma represents the manager home screen, designed for low complexity use. Mb represents the Manager's screen when trying to broadcast a message to all employees. These screens are not used directly in the Use Cases mentioned but they can be affected from actions committed in any use case.

From our initial mockup we have improved the Ma screen by removing distracting buttons or unnecessary information. Before, in the initial mockup, the manager had a button to reply to messages and had a table with multiple pieces of information about the message they received. We have eliminated these and made the messages straightforward by just printing the employee, location, and issue in a concise sentence. Instead of including a time stamp, we update the messages with most recent messages at the top. We have also moved the Create New Message Button to the bottom left of the screen to reduce distractions on the screen.

Figure Ma: Manager Home Screen

Figure Mb: Manager Screen when Broadcasting a Message

Design of Tests

Changes we made in Design of Tests:
Removed: MANAGER - managerMessageListener, managerEventListener, managerMessageSender
HOST - hostMessageListener,hostEventListener, hostMessageSender, UpdateTable, orderTableList, orderTableList
OWNER - addEmployee, removeEmployee, changeEmployeePosition, addMenuItem, removeMenuItem, changePrice, logIn, logOut (took out everything from Owner)
WAITER - openTicket, printTicket, paid, waiterMessageListener, waiterEventListener, waiterMessageSender
MESSAGE CONTROLLER - messageListener, messageParser, sendMessage, sendTicket, ticketListener, addTabAddress, removeTabAddress
CHEF - addIngredientToInventory, removeIngredientFromInventory, addTicketToQueue, orderTicketQ, changeDish, chefMessageListener, chefEventListener, chefMessageSender, changeTicketColor

Added: MANAGER - logOut, deleteMessage, sendmassnotification, addmessagetolist
HOST - loadTables, notifyManager, notifyWaiter, seat, sendSeated, logOut, paid, cleaned, loadWaiters
OWNER - nothing
WAITER - logOut, loadMenu, addGCorCoupon, addDishToTicket, removeDishFromTicket, addComment, notifyManager, sendTicket, paid, notifyWaiter, removeLowInventoryDishes
MESSAGE CONTROLLER - addWaiterSender, addHostSender, addChefSender, addManagerSender, removeWaiterSocket, removeHostSocket, removeChefSocket, removeManagerSocket
CHEF - chefTicketListener, changeTicketLocation, setUpMessageController, openTicketScreens, backToMainScreen, logOut, notifyManager, addNotification, updateScreen

Manager

Test Case Identifier: TC-01 **Function Tested:** logOut(): boolean **Pass/Fail Criteria:** Test will pass if the manager successfully logs out of the system.	
Test Procedure	**Expected Results**
Call Function (Pass)	Logs the manager out of the system and returns true.

Call Function (Fail)	Does not log the manager out of the system and returns false.

Test Case Identifier: TC-02
Function Tested: deleteMessage(int index): boolean
Pass/Fail Criteria: Test will pass if the message is successfully deleted.

Test Procedure	Expected Results
Call Function (Pass)	Deletes the message at a given index in the manager's message list and returns true.
Call Function (Fail)	Does not delete the message and returns false.

Test Case Identifier: TC-03
Function Tested: sendMassNotification(String content): boolean
Pass/Fail Criteria: Test will pass if the notification is successfully sent to all employees.

Test Procedure	Expected Results
Call Function (Pass)	Notification is successfully sent to all employees and it returns true.
Call Function (Fail)	Notification does not get sent and returns false.

Test Case Identifier: TC-04
Function Tested: addMessageToList(Message m): boolean
Pass/Fail Criteria: Test will pass if the new message for the manager is added to the list of messages and displayed on the screen.

Test Procedure	Expected Results
Call Function (Pass)	New message for the manager is displayed and it returns true.
Call Function (Fail)	New message for the manager is not displayed and it returns false.

Chef

Test Case Identifier: TC-05
Function Tested: chefTicketListener(Ticket ticket): boolean
Pass/Fail Criteria: Test will pass if the incoming tickets from the waiters are read and placed in the queue that the chef can view.

Test Procedure	Expected Results
Call Function (Pass)	The incoming tickets are read, displayed on the chef's screen and the function returns true.
Call Function (Fail)	The incoming tickets are not read and the function returns false.

Test Case Identifier: TC-06
Function Tested: changeTicketLocation(char oldstatus, Ticket t): boolean
Pass/Fail Criteria: Test will pass if the location of the ticket changes based on the status change.

Test Procedure	Expected Results
Call Function (Pass)	The location of the ticket changes and it returns true.
Call Function (Fail)	The location of the ticket doesn't change and it returns false.

Test Case Identifier: TC-07
Function Tested: setUpMessageController(): boolean
Pass/Fail Criteria: Test will pass if the message controller is set up.

Test Procedure	Expected Results
Call Function (Pass)	The message controller is set up and it returns true.
Call Function (Fail)	The message controller is not set up and it returns false.

Test Case Identifier: TC-08
Function Tested: openTicketScreens(long ticketNumber): boolean

Pass/Fail Criteria: Test will pass if the contents of the ticket are displayed on the chef's screen.	
Test Procedure	**Expected Results**
Call Function (Pass)	The contents of the ticket are displayed on the chef's screen and it returns true.
Call Function (Fail)	The contents of the tickets are not displayed and it returns false.

Test Case Identifier: TC-09 **Function Tested:** backToMainScreen(): boolean **Pass/Fail Criteria:** Test will pass if the chef's screen goes back to the main page showing the ticket queue.	
Test Procedure	**Expected Results**
Call Function (Pass)	The chef's screen goes back to the main page and it returns true.
Call Function (Fail)	The chef's screen doesn't go back to the main page and it returns false.

Test Case Identifier: TC-10 **Function Tested:** logOut(): boolean **Pass/Fail Criteria:** Test will pass if chef successfully logs out of the system.	
Test Procedure	**Expected Results**
Call Function (Pass)	The chef successfully logs out of the system and it returns true.
Call Function (Fail)	The chef doesn't log out of the system and it returns false.

Test Case Identifier: TC-11 **Function Tested:** notifyManager(): boolean **Pass/Fail Criteria:** Test will pass if a notification is sent to the manager.	
Test Procedure	**Expected Results**
Call Function (Pass)	A notification is sent to the manager and it returns

	true.
Call Function (Fail)	A notification doesn't get sent to the manager and it returns false.

Test Case Identifier: TC-12 **Function Tested:** addNotification(String message): boolean **Pass/Fail Criteria:** Test will pass if a notification gets added to the chef's screen.	
Test Procedure	**Expected Results**
Call Function (Pass)	A notification gets added to the chef's screen and it returns true.
Call Function (Fail)	A notification doesn't get added to the chef's screen and it returns false.

Test Case Identifier: TC-13 **Function Tested:** updateScreen(): boolean **Pass/Fail Criteria:** Test will pass if the chef's screen successfully updates.	
Test Procedure	**Expected Results**
Call Function (Pass)	The chef's screen successfully updates and it returns true.
Call Function (Fail)	The chef's screen doesn't update and it returns false.

Waiter

Test Case Identifier: TC-14 **Function Tested:** logOut(): boolean **Pass/Fail Criteria:** Test will pass if the waiter successfully logs out of the system.	
Test Procedure	**Expected Results**
Call Function (Pass)	Waiter will be logged out of the system and the function will return true.
Call Function (Fail)	Waiter will not be able to log out of the system and the function will return false.

Test Case Identifier: TC-15
Function Tested: loadMenu(): boolean
Pass/Fail Criteria: Test will pass if the menu is successfully loaded from Database C.

Test Procedure	Expected Results
Call Function (Pass)	Waiter will be able to access the menu on their tablet and it will return true.
Call Function (Fail)	Waiter will not be able to see the menu on their tablet and it will return false.

Test Case Identifier: TC-16
Function Tested: addGCorCoupon(double price): boolean
Pass/Fail Criteria: Test will pass if the waiter can successfully add a gift card or coupon to the table's ticket.

Test Procedure	Expected Results
Call Function (Pass)	Waiter can add a gift card or coupon to the table's ticket and it will return true.
Call Function (Fail)	Waiter will not be able to add a gift card or coupon to the table's ticket and it will return false.

Test Case Identifier: TC-17
Function Tested: addDishToTicket(Dish dish): boolean
Pass/Fail Criteria: Test will pass if the waiter can successfully add a dish to their table's ticket.

Test Procedure	Expected Results
Call Function (Pass)	Waiter can add a dish to their table's ticket and it will return true.
Call Function (Fail)	Waiter will not be able to add a dish to their table's ticket and it will return false.

Test Case Identifier: TC-18
Function Tested: removeDishFromTicket(int indexInTicket): boolean
Pass/Fail Criteria: Test will pass if the waiter can successfully remove a dish from their table's ticket.

Test Procedure	Expected Results
Call Function (Pass)	Waiter can remove a dish from their table's ticket and it will return true.
Call Function (Fail)	Waiter will not be able to remove a dish from their table's ticket and it will return false.

Test Case Identifier: TC-19
Function Tested: addComment(int ind, String com): boolean
Pass/Fail Criteria: Test will pass if the waiter can add comments to a specific dish on their table's ticket.

Test Procedure	Expected Results
Call Function (Pass)	Waiter can add comments on a dish from their table's ticket and it will return true.
Call Function (Fail)	Waiter will not be able to add comments on a dish from their table's ticket and it will return false.

Test Case Identifier: TC-20
Function Tested: notifyManager(Ticket currTicket2, String message): boolean
Pass/Fail Criteria: Test will pass if the waiter can successfully send the manager a notification that a specific table needs them.

Test Procedure	Expected Results
Call Function (Pass)	Waiter can send the manager a notification with the specific table number that needs the manager's attention and it will return true.
Call Function (Fail)	Waiter will not be able to send the manager a notification and it will return false.

Test Case Identifier: TC-21
Function Tested: sendTicket(Ticket t): boolean
Pass/Fail Criteria: Test will pass if the waiter can successfully send a table's ticket to the chef.

Test Procedure	Expected Results

Call Function (Pass)	Chef will receive a table's ticket with the correct time it was sent and the function will return true.
Call Function (Fail)	Chef will not receive a table's ticket and the function will return false.

Test Case Identifier: TC-22
Function Tested: paid(int tableNumber): boolean
Pass/Fail Criteria: Test will pass if waiter can mark the table as paid and the host receives an alert that the table has paid.

Test Procedure	Expected Results
Call Function (Pass)	Waiter can mark the table as paid so the host can see it has been paid and can seat it again, the function returns true.
Call Function (Fail)	Waiter will not be able to mark the table as paid and the function will return false.

Test Case Identifier: TC-23
Function Tested: notifyWaiter(int tableNumber): boolean
Pass/Fail Criteria: Test will pass if a waiter can send another waiter a notification that their table is requesting them.

Test Procedure	Expected Results
Call Function (Pass)	The waiter can notify another waiter that their table is requesting them, the function will return true.
Call Function (Fail)	The waiter will not be able to notify another waiter that their table is requesting them, the function will return false.

Test Case Identifier: TC-24
Function Tested: removeLowInventoryDishes(String[ ] dishes): boolean
Pass/Fail Criteria: Test will pass if a certain dish gets removed from the menu when the chef can no longer cook them due to low inventory.

Test Procedure	Expected Results

Call Function (Pass)	Waiter can no longer see the dish that cannot be served and the function will return true.
Call Function (Fail)	Waiter will still be able to see the dish even though it cannot be prepared and the function will return false.

Host

Test Case Identifier: TC-25
Function Tested: loadTables(): boolean
Pass/Fail Criteria: Test will pass if it successfully loads the tables from Database B.

Test Procedure	Expected Results
Call Function (Pass)	It loads the tables from Database B and returns true.
Call Function (Fail)	It does not load the tables from Database B and returns false.

Test Case Identifier: TC-26
Function Tested: notifyManager(): boolean
Pass/Fail Criteria: Test will pass if the manager successfully receives the notification from the host.

Test Procedure	Expected Results
Call Function (Pass)	The notification successfully gets sent to the manager and it returns true.
Call Function (Fail)	The notification is not sent to the manager and it returns false.

Test Case Identifier: TC-27
Function Tested: notifyWaiter(Message m): boolean
Pass/Fail Criteria: Test will pass if the waiter successfully receives the message from the host.

Test Procedure	Expected Results
Call Function (Pass)	The notification successfully gets sent to the waiter and it returns true.

Call Function (Fail)	The notification is not sent to the waiter and it returns false.

Test Case Identifier: TC-28
Function Tested: seat(String waiterName, int tableNumber): int
Pass/Fail Criteria: Test will pass if a ready table with the corresponding table number is seated with the correct waiter.

Test Procedure	Expected Results
Call Function (Pass)	The table with the corresponding table number is seated with the correct waiter and it returns 0.
Call Function (Fail)	If the table is not ready to be sat, it returns -1.

Test Case Identifier: TC-29
Function Tested: sendSeated(long waiterId, int tableNumber): boolean
Pass/Fail Criteria: Test will pass if host successfully send a message to the waiter telling them that they were seated a table.

Test Procedure	Expected Results
Call Function (Pass)	The waiter receives a message from the host notifying them that they were sat a table and it returns true.
Call Function (Fail)	The waiter does not receive a message from the host and it returns false.

Test Case Identifier: TC-30
Function Tested: logOut(): boolean
Pass/Fail Criteria: Test will pass if the host successfully logs out of the system.

Test Procedure	Expected Results
Call Function (Pass)	The host successfully logs out of the system and the function returns true.
Call Function (Fail)	The host does not log out of the system successfully and the function returns false.

Test Case Identifier: TC-31
Function Tested: paid(int tableNumber): boolean
Pass/Fail Criteria: Test will pass if a seated table on the host screen successfully updates to paid when the waiter marks the table as paid.

Test Procedure	Expected Results
Call Function (Pass)	Host screen will successfully update a seated table as paid and return true.
Call Function (Fail)	Host screen will not update successfully and return false.

Test Case Identifier: TC-32
Function Tested: cleaned(int tableNumber): boolean
Pass/Fail Criteria: Test will pass if host screen updates a paid table that was just cleaned into the ready list.

Test Procedure	Expected Results
Call Function (Pass)	Host screen will successfully add the cleaned table into the ready list and return true.
Call Function (Fail)	Host screen will not successfully add the cleaned table to the ready list and return false.

Test Case Identifier: TC-33
Function Tested: loadWaiters(): boolean
Pass/Fail Criteria: Test will pass if the host can see all the waiters that are currently logged in to the system on their screen.

Test Procedure	Expected Results
Call Function (Pass)	Host will be able to see all the waiters that are currently working on their screen and it returns true.
Call Function (Fail)	Host will not be able to see the list of waiters and it will return false.

Message Controller

Test Case Identifier: TC-34
Function Tested: addWaiterSender(long id, MessageControllerSender sender): boolean
Pass/Fail Criteria: Test will pass if the waiter is added to the list of output streams so that the message controller has a way of communicating with the waiter.

Test Procedure	Expected Results
Call Function (Pass)	The waiter is added to the list of output streams and it returns true.
Call Function (Fail)	The waiter is not added to the list of output streams and it returns false.

Test Case Identifier: TC-35
Function Tested: addHostSender(long id, MessageControllerSender sender): boolean
Pass/Fail Criteria: Test will pass if the host is added to the list of output streams so that the message controller has a way of communicating with the host.

Test Procedure	Expected Results
Call Function (Pass)	The host is added to the list of output streams and it returns true.
Call Function (Fail)	The host is not added to the list of output streams and it returns false.

Test Case Identifier: TC-36
Function Tested: addChefSender(long id, MessageControllerSender sender): boolean
Pass/Fail Criteria: Test will pass if the chef is added to the list of output streams so that the message controller has a way of communicating with the chef.

Test Procedure	Expected Results
Call Function (Pass)	The chef is added to the list of output streams and it returns true.
Call Function (Fail)	The chef is not added to the list of output streams and it returns false.

Test Case Identifier: TC-37
Function Tested: addManagerSender(long id, MessageControllerSender sender): boolean
Pass/Fail Criteria: Test will pass if the manager is added to the list of output streams so that the message controller has a way of communicating with the manager.

Test Procedure	Expected Results
Call Function (Pass)	The manager is added to the list of output streams and it returns true.
Call Function (Fail)	The manager is not added to the list of output streams and it returns false.

Test Case Identifier: TC-38
Function Tested: removeWaiterSocket(long id): boolean
Pass/Fail Criteria: Test will pass if the waiter is removed from the list of output streams.

Test Procedure	Expected Results
Call Function (Pass)	The waiter is removed from the list of output streams and it returns true.
Call Function (Fail)	The waiter is not removed from the list of output streams and it returns false.

Test Case Identifier: TC-39
Function Tested: removeHostSocket(long id): boolean
Pass/Fail Criteria: Test will pass if the host is removed from the list of output streams.

Test Procedure	Expected Results
Call Function (Pass)	The host is removed from the list of output streams and it returns true.
Call Function (Fail)	The host is not removed from the list of output streams and it returns false.

Test Case Identifier: TC-40
Function Tested: removeChefSocket(long id): boolean
Pass/Fail Criteria: Test will pass if the chef is removed from the list of output streams.

Test Procedure	Expected Results

Call Function (Pass)	The chef is removed from the list of output streams and it returns true.
Call Function (Fail)	The chef is not removed from the list of output streams and it returns false.

Test Case Identifier: TC-41 **Function Tested:** removeManagerSocket(long id): boolean **Pass/Fail Criteria:** Test will pass if the manager is removed from the list of output streams.	
Test Procedure	**Expected Results**
Call Function (Pass)	The manager is removed from the list of output streams and it returns true.
Call Function (Fail)	The manager is not removed from the list of output streams and it returns false.

Global Control Flow

Execution Orderliness

Our application, *Serve with Ease* is an event driven software. Every action is a consequence of another action. For example, the waiter does not greet a table until the host updates the table status to seated for that table and the chef doesn't start any dishes until he receives the ticket order from the host. The subsystems use interrupts to determine when to prompt for/execute the next action rather than polling. In this way, the amount of time wasted checking to see if an action needs to be completed is eliminated and thus making our system the most efficient.

Time Dependency

Our application, *Serve with Ease* is completely event response driven and does not depend on real-time. All actions are prompted as a result of a previously executed action so there is no need for timers. The system is not periodic as it does not check on a regular basis to see if any actions need to be taken. For example, the system does not check continuously for seated tables to alert the waiter but rather waits for a message from the host interface, then alerts the waiter through that message. Our application is time independent making it faster as it does not have to spend time waiting for events to occur.

Concurrency

Serve with Ease contains multiple subsystems that all connect to one server. This requires multithreading. Multiple interfaces will access the server at the same time so it is necessary to use concurrency. We plan to have individual threads handling each client. As multiple clients will be accessing the databases, synchronization is needed to ensure the reliability of data. It is important when one client is accessing or modifying certain data that other clients do not have access to it at the same time. Synchronization is also needed to prevent race conditions.

Hardware Requirement

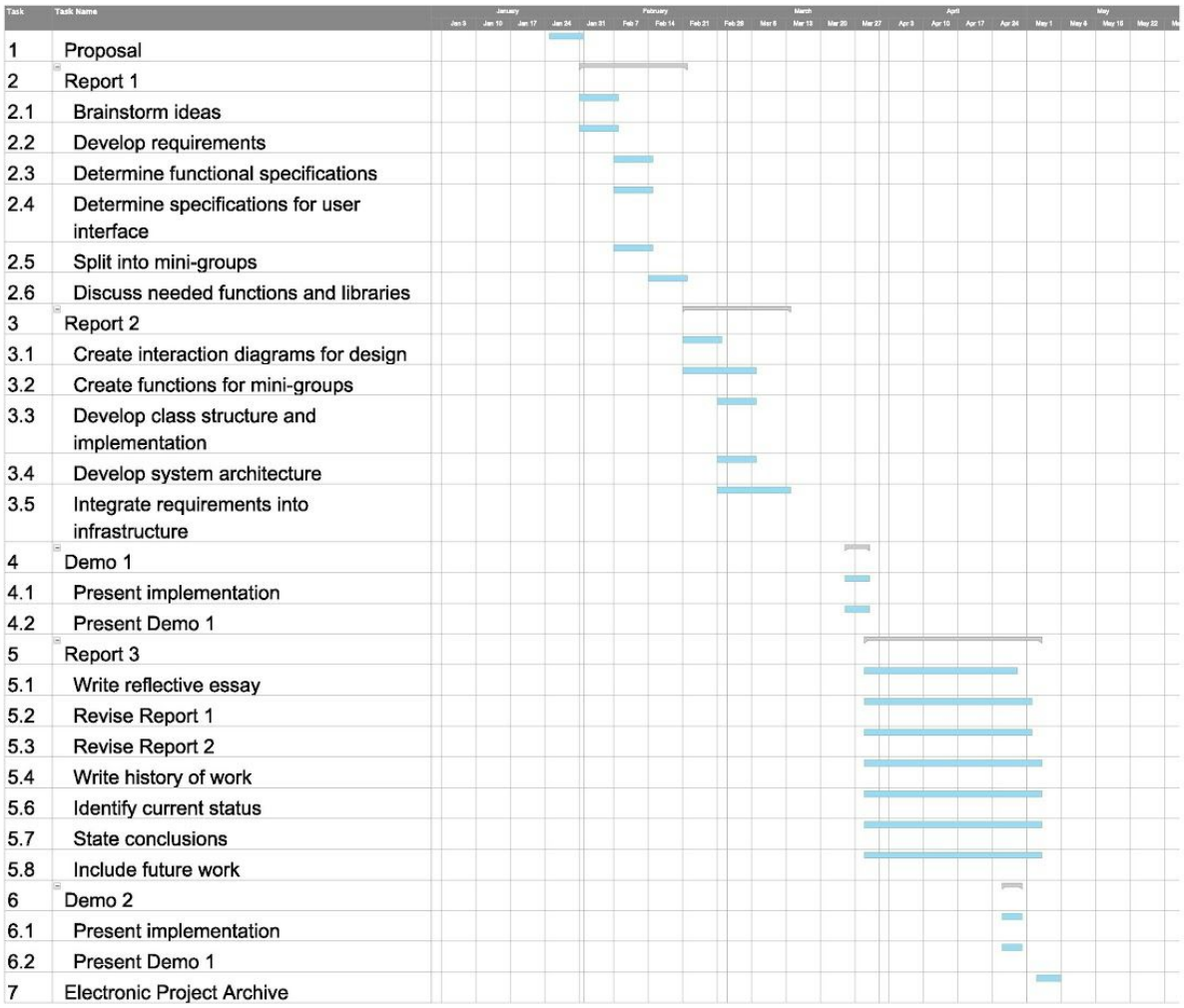
In order to use our program to its highest potential, users need an Android Tablet. This will allow them to use all of the functions of our software with ease while also being mobile. To run our software, the hardware requirements are:

Samsung Tablet		
Hardware Component	Minimum Requirements	Recommended Requirements
Processor	Qualcomm APQ 8016	Qualcomm Snapdragon 801QuadCore
RAM	1.5GB	3GB
Hard Drive Space	16GB	32GB
Screen Size	9.6 in	8 in
Resolution	1280x800	1920x1200

Project Management & History of Work

Gantt Chart

The objectives of our project throughout the duration of the semester have been depicted in the below Gantt Chart. Each task in the Gantt Chart is a milestone in the project, and each decimal numbered task is a subtask of the main task below. The chart is updated to the best of our knowledge in regard to the responsibilities and deadlines we have been given thus far and is subject to change.



Task	Task Name
1	Proposal
2	Report 1
2.1	Brainstorm ideas
2.2	Develop requirements
2.3	Determine functional specifications
2.4	Determine specifications for user interface
2.5	Split into mini-groups
2.6	Discuss needed functions and libraries
3	Report 2
3.1	Create interaction diagrams for design
3.2	Create functions for mini-groups
3.3	Develop class structure and implementation
3.4	Develop system architecture
3.5	Integrate requirements into infrastructure
4	Demo 1
4.1	Present implementation
4.2	Present Demo 1
5	Report 3
5.1	Write reflective essay
5.2	Revise Report 1
5.3	Revise Report 2
5.4	Write history of work
5.6	Identify current status
5.7	State conclusions
5.8	Include future work
6	Demo 2
6.1	Present implementation
6.2	Present Demo 1
7	Electronic Project Archive

History of Work

Merging and Collecting Contributions

Every week, we updated our group Google Drive specifying the responsibilities that need to be accomplished. When we met in person at least twice throughout the week, we collectively delegated tasks among the group and agreed to have these tasks completed at a given time prior to the deadline. Then when each team member completed her portion, the finished product was inserted into the report on a Google Doc. Once all of the parts were collected, a team member would ensure the formatting was consistent throughout the report and ready for submission.

Problems

Initially, it was difficult to find a time to meet which accommodated everyone's schedule. We discussed which times and days of the week worked best for everyone and ultimately, found times to meet regularly throughout the week. Also, if any member in the group struggled with

completing their portion in a timely manner, then the other group members would remind the team member and provide help with the work as needed. Another issue that came up was managing the code on various builds of laptops. We were able to resolve this by utilizing GitHub to keep track of updated versions of our application code, and all of us used Eclipse software to work on the code. This way, we avoided any issues of using different software platforms. Overall though, we successfully overcame the issues we had.

History of Work

January 26th – January 31st

When the course began, we quickly came together as a six-person group and began brainstorming ideas. We decided to work on the Restaurant Automation project as a group, while having a few group members wielding hands-on restaurant experience, and then we started brainstorming ideas to include in our project. We decided to develop an application to facilitate employee responsibilities and improve customer satisfaction. Once we decided which ideas we wanted to include in our application, we wrote up a Proposal of our idea.

February 1st – February 21st

After receiving feedback on our Proposal, we went ahead and began working on Report 1. We delegated portions of each part of Report 1 to each respective team member. We met as a group several times throughout the week, and ensured each member was carrying her fair share of work consistently.

February 22nd – March 13th

We continued working on Report 2 as a group. Similar to Report 1, we split up tasks for each team member and then combined our completed portions together. We have completed the technical specifications of our application, and for Report 2, we are working on the design component of the application. We are also working on the development of the application.

March 13th – March 25th

The team is currently in the process of developing the Graphical User Interface (GUI) and database for our application. We completed the primary development and user interface of our application in preparation for Demo 1. We are working on different portions of the project simultaneously. We just completed Report 2, and Demo 1 is coming up soon.

March 26th – April 26th

We are currently in the process of wrapping up Report 3 in preparation for Demo 2. A lot of the documentation we have is merely being updated and compiled together to create Report 3. We have also been making some modifications for the application. The client/server functionality now works in real-time, among several new additions.

Future Status

In the future, after Demo 2 is complete, we will reflect on our project and how we worked together as a team. We will also seek out improvements for our project and consider future additional components we can add to our application. We hope that our application can be scaled such that it may be used by larger restaurants, prominent fast food chains, fine dining services, and various other areas in industry.

Key Accomplishments

Below are some accomplishments that we have had throughout the project. These include things we have learned about software (in code) as well as working in teams or on projects (outside of code).

Key Accomplishments **outside of code**:

- Collaborated with a group of six individuals to deliver a working application.
- Provided clear technical documentation throughout duration of the project.
- Evaluated each individual's key strengths to allocate tasks efficiently and work effectively as a group.
- Developed a better understanding of test-driven software development and Agile methodologies.
- Actively worked to improve the internal system throughout the semester.
- Able to accept critical feedback and make changes accordingly.
- Spoke with actually employees for in depth understanding of client needs.
- Listened to client needs and provided solutions accordingly.
- As individual developers, we all have a better understanding of how to provide for user driven needs.
- As individual developers, we all have a better understanding of the importance of ‘out-of-box’ solutions.
- As individual developers, we all have a better understanding of how to adapt to the skills, coding style, thoughts, and work ethics of other team members.
- As taught in class, we have a better understanding of how to improve and innovate on existing ideas.

Key Accomplishments **in code**:

- Created an application to work on multiple devices at once.
- Successfully implemented a distributed database.
- Developed an easy and instantaneous messaging system to communicate between employees.

- Real time updating of Host home screen upon Waiter action
- Split Host's home screen into two separate arrays and maintain active interaction between the two.
- Hanging notification for 'Recently Sat' alert
- 'Recently Sat' and 'Hot Food' alerts on table button on Waiter home screen
- Effectively used new data structures such as Hashmaps and Arraylists

Future Work

We see *Serve With Ease* as an application that can be scaled to not only sit down restaurants, but also fast food joints. It can be used by chain restaurants so multiple restaurants of the same chain around the country can link up their applications and keep track of sales altogether and the common menu. SWE can be used for types of food stores — restaurants, ice cream parlors, fast food joints, food courts, etc. The application can be expanded to different prototypes to provide a layout for different stores, such as the ones listed above. Since the workers at an ice cream parlor and a fast food joint will not have the same needs or roles as workers at a restaurant, we need to anticipate such changes. Creating multiple prototypes of our application will allow our clients to choose which one is best suited for their needs.

For our base case of working with restaurants we would like to continue to improve our application further. Specifically, we would like to do the following in the future:

<u>Owner</u>

In the future we plan to add an owner screen where the owner can:

1. change the price of menu items
2. add and remove employees
3. change inventory thresholds and current amounts

All of these actions will be activated with buttons. Right now all of these things are done through user-friendly text files.

<u>Host</u>

In the future we plan to:

1. Add a timestamp to each table on the Host's screen. This will be updated every time the table is sat. It will hold the time the table was last sat. This can be helpful when figuring out
2. Save the status of each table into database C. This should be updated everytime the table changes its status on the host screen. This will allow the host to pull not only the table numbers but the statuses from the database when he or she logs on. So if during the shift one host logs off and another logs on, the statuses of all the tables will be saved.

Waiter

In the future we plan to:

1. Add a duplicate dish button. On the open ticket screen, we plan to add a duplicate dish button that will allow the server to select a dish that they added to the ticket already and duplicate it. This will duplicate not only the name of the dish but all of the comments under it.
2. A split check feature when printing the check. This will allow the waiter to split the amount of the bill into different checks in the case that a table wants to pay separately. To implement this we will have the each dish have a field that holds the seat number. So each time the waiter adds an item to the ticket they have to add the seat number of the customer who ordered it. Then when the table wants to split the check they can do it by seat number.
3. Implement a shared ticket option. If a large party comes in, usually they will require more than one waiter to take care of them. When a host seats them, they will need to select the two waiters that have that table and seat them there. Then the ticket for that table will be shared and accessible by both the waiters. To implement this we plan to use shared memory along with a mutex for each of the tickets that are shared. This mutex will be used to allow only one waiter to access the ticket (and add dishes to it) at a time. This will prevent any race conditions between the two waiters who are sharing the table and ticket.

Manager

In the future we plan to:

1. Be able to send message to particular position not just broadcast. The manager will be able to create a message that can be sent to only selected employees. These selected employees will be chosen from a list of all of the employees who are currently logged in.
2. Change menu that servers see. A manager will be able to add or remove a dish from the menu that the waiters see on their tablet and order from. This will allow for managers to place dishes that are specials of the day or the week on the menu so the waiter can still place an order when a customer wishes to order a special.
3. View statistics about revenues, dishes, and waiters. The expansion here seems limitless. Given the data the system collects when the waiter sends a paid ticket to the database, the system can formulate a nice statistical print out on the manager screen. These statistics could be how much money the restaurant made yesterday and today, or a year ago. It could also let the manager know what the most and least ordered (popular) dishes were this week. We can also expand to waiter's and see what waiter has the highest

revenue/customer ratio to see who is selling the most. All of these can help managers make better and more profitable choices about the restaurant.

Chef

In the future we plan to:

1. Implement a clock on the screen so the Chef is able to view the time easily when needed.
2. Display a timestamp on each ticket so the Chef can know when the ticket was ordered.
3. Implement the hot food notification to a waiter. This will send a hot food message to the waiter once their ticket is completed, letting the waiter know it is time to pick up his food for his table.
4. Recognize priority tickets. When a waiter sends a priority ticket to the chef, the ticket will be placed at the front of the list and be shown in a distinct color.
5. Decrement dish to maintain inventory records. The menu, uploaded in by the owner or manager, should contain the amount of ingredients per dish, as listed by the chef. When the chef changes the status of a ticket from unstarted to started, the corresponding amount of each of the item on the ticket should be decremented in the inventory. This will allow for restaurants to keep track of stock more accurately. This can be expanded to tracking the most commonly used items in the kitchen as well as the least commonly used.

Overall, we anticipate *Serve With Ease* to be a user friendly application, adaptable to various restaurant styles.

References

Previous Group Projects:

- Group 2 Spring 2012
- Group 4 Spring 2014
- Group 3 Spring 2015
- Group 11 Spring 2012

Employee References:

- Emma Roussos: Server/Hostess at Colonial Diner
- Christina Segerholm: Server/Hostess at Carrabba's Italian Grille
- Sally Kobuta : Manager at Carrabba's Italian Grille
- Danielle Segerholm: Hostess at Carrabba's Italian Grille

http://www.ece.rutgers.edu/~marsic/books/SE/book-SE_marsic.pdf
https://en.wikipedia.org/wiki/Technical_documentation
https://en.wikipedia.org/wiki/Unit_testing
https://en.wikipedia.org/wiki/Integration_testing
http://www.doc-department.com/what-are-user-documentation-and-technical-documentation/
https://try.github.io/levels/1/challenges/1
http://www.tutorialspoint.com/java/java_networking.htm
https://en.wikipedia.org/wiki/Gson
http://www.tutorialspoint.com/swing/
http://www.dreamincode.net/forums/topic/225444-i-need-to-create-an-on-screen-keyboard-gui-using-java/