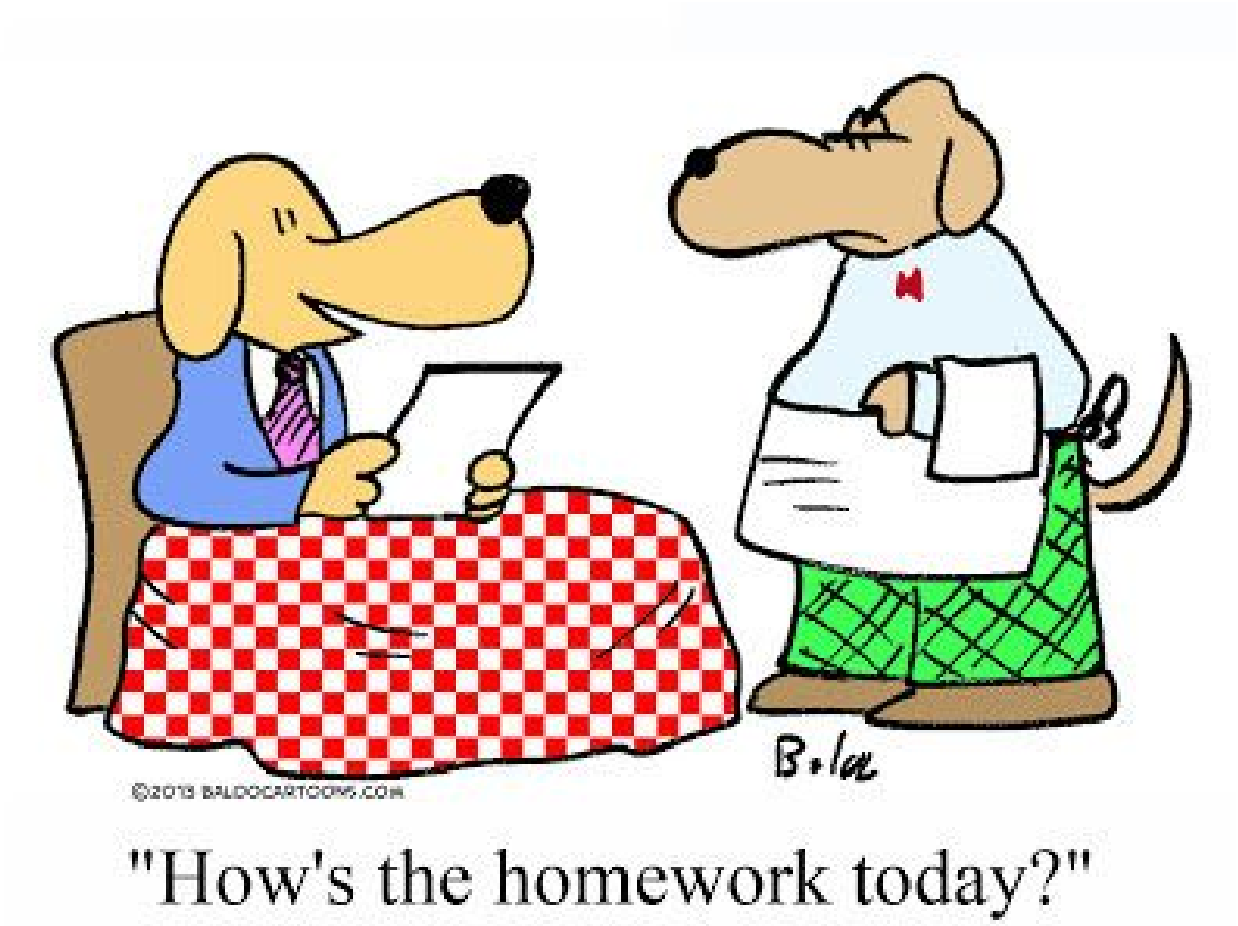
SWE: Serve with Ease

GROUP #1

Project: Restauranting Made Easy

Report 2



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: Use Case 1 Interaction Diagram - 50%, Architectural Style-33%, Mapping Subsystems to Hardware-50%, (Persistent Data Storage, Network Protocol, Execution Orderings, Time Dependency, Concurrency, Identifying Subsystems, Data Structures)-50 %
- . software design:
- . coding:
- . debugging:
- . report preparation: Report 2 Part 1 Formatting, Part 2 Formatting, Full Report Formatting
- . Other: Overall revisions

Annie:

- . requirements specification: Use Case 2 Interaction Diagram - 50%, (Class Diagram, Class Descriptions, Data Types & Operation Signatures) 33%, Design Test Cases- 33%
- . software design:
- . coding:
- . debugging:
- . report preparation: organization
- . Other: : Overall Revisions

Christina S:

- . requirements specification: Use Case 1 Interaction Diagram - 50%, (Class Diagram, Class Descriptions, Data Types & Operation Signatures) 33%, (Persistent Data Storage, Network Protocol, Execution Orderings, Time Dependency, Concurrency)-50%, Algorithms - 100%
- . software design: Designed overall layout of system and architecture, creator of distributed databases and message controller, Major Ideas behind System Design and Database distribution
- . coding: Java Classes - started Table, DishData, Dish, Ticket, interface classes, Message, event classes
- . debugging:
- . report preparation: Report 2 Part 1 Formatting, Full Report Formatting
- . Other: Setup template and instructions for rest of group to use, it described how to go about making a interaction diagram, Revised each sub groups interaction diagram and caption., Overall Revisions

Emma:

- . requirements specification: Use Case 3 Interaction Diagram - 100%, Architectural Style-33%, Mapping Subsystems to Hardware-50%, Identifying Subsystems,, Design Test Cases- 33%
- . software design:
- . coding:
- . debugging:
- . report preparation:
- . Other: : Overall Revisions

Christina P:

- . requirements specification: Revised Use Case 3 Caption and Design Principles- 100%, Architectural Style-33%, Traceability Matrix 100%, History of Work- 100%, Design Test Cases- 33%
- . software design:
- . coding:
- . debugging:
- . report preparation:Report 2 Part 1 Formatting
- . Other: : Overall Revisions

Nishtha:

- . requirements specification: Use Case 2 Interaction Diagram - 50%,(Class Diagram, Class Descriptions, Data Types & Operation Signatures) 33%, Data Structures-50%
- . software design:
- . coding:
- . debugging:
- . report preparation:Report 2 Part 1 Formatting, organization
- . Other: : Overall Revisions

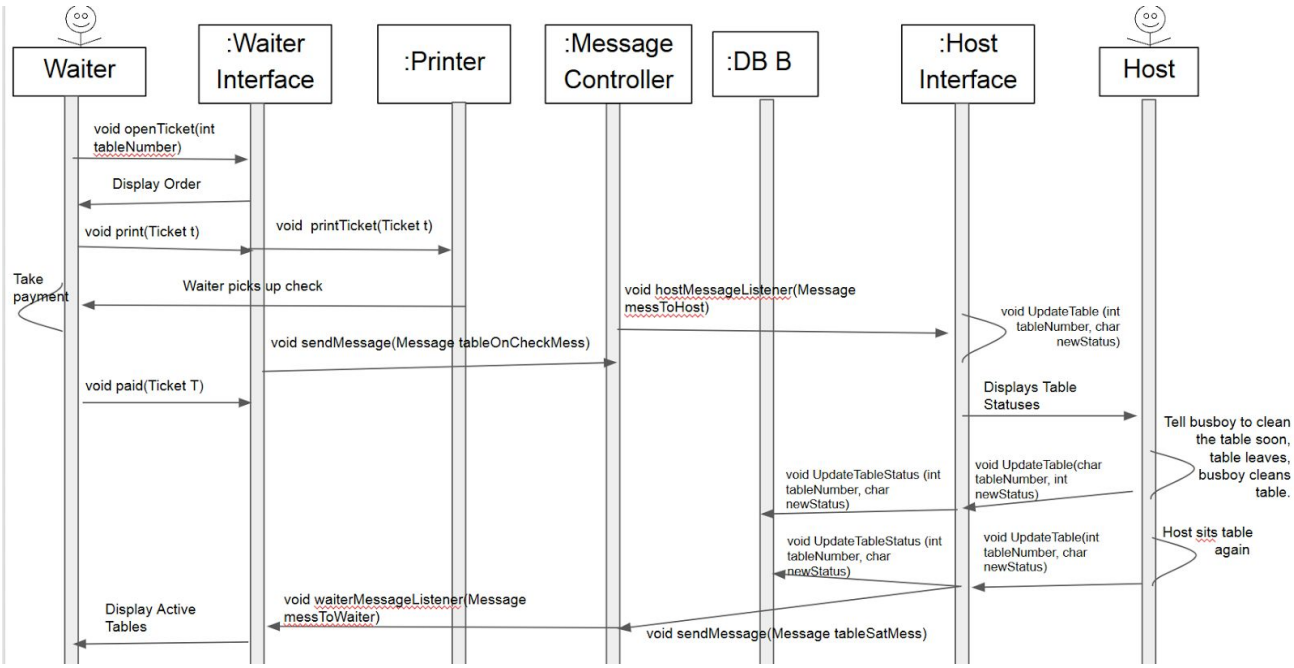
Summary of Changes

- Changed name for Service for Servers to Serve with Ease (SWE)
- Waiter Interface: Removed specific modification buttons now will just have modifications typed as a comment.
- Modified database controllers:
    - Database A Controller controls data in Database A (holds log-in and menu data).
    - Host Interface controls data in Database B (holds Table statuses).
    - Chef Interface controls data in Database C(holds Ticket Queue and inventory).
- Added message listener and message sender for each interface.
- Broke Host screen up to display two lists of tables instead of one. One list for ready tables and one for non ready tables.
- Interviewed two employees at Carrabba's Italian Grille. Added their input into Report 1 problem statement.

Table of Contents:

Interaction Diagrams

Use Case 1:



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. 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.

Design Principles

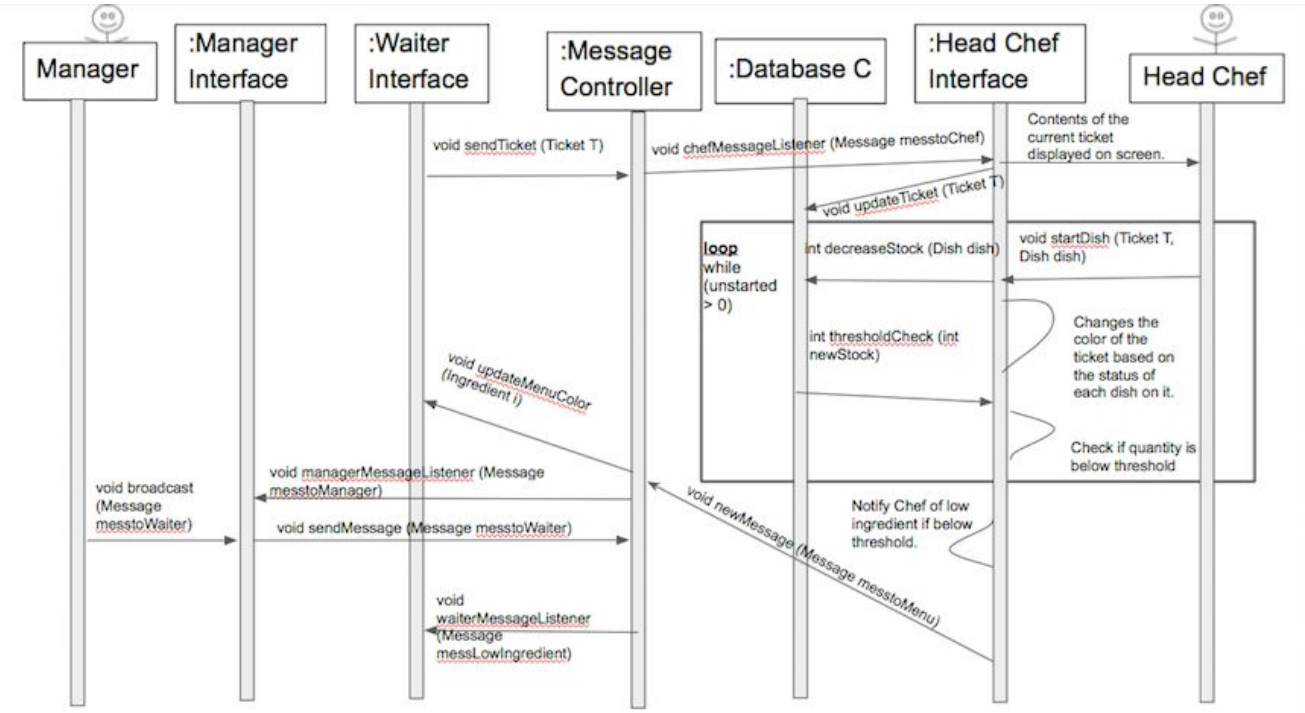
1. The Expert Doer principle is implemented by the:
    a. Printer. The printer knows how to print a ticket properly given a ticket and so handles this task.
    b. Message Controller. The message controller is able to decode a message in order it to forward it to the correct destination. Instead of 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.

Use Case 2:

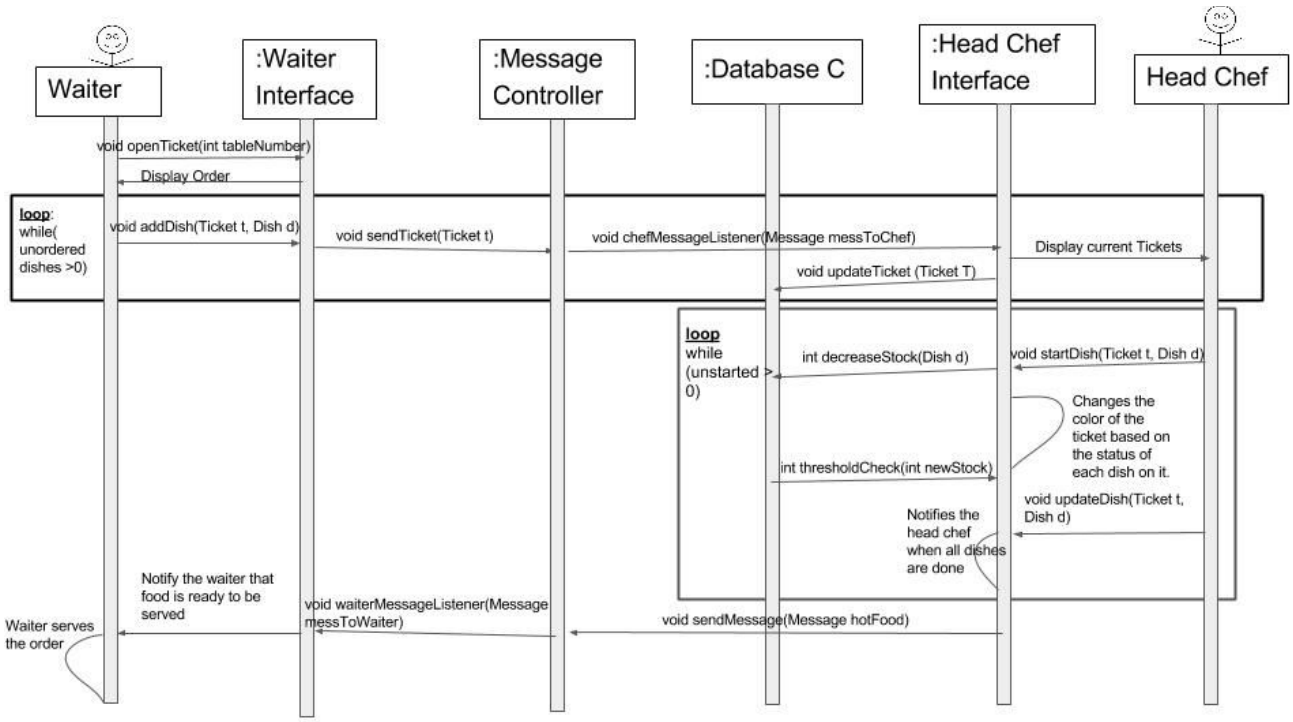


The waiter first starts the process by sending the ticket to the head chef. The ticket will get added to the queue and the color will be red to indicate that none of the dishes have been started. Then the chef chooses to click the ticket and, within that, the chef will click on a dish in order to start the process of making it. Based on the dish started, the appropriate amount of ingredients used will be deducted from the stock in the inventory. This occurs every time any dish is started. Additionally, the specified threshold of the inventory will be checked upon each decrease. If the stock in the inventory falls below the threshold, then a notification is sent to the chef, the manager, and the color of the menu is updated. From there, the manager can choose whether to broadcast a message to the waiters to inform them of the low stock. This process will start again when a waiter sends in another ticket.

Design Principles:

1. Expert Doer 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, ingredients, and the current orders and has no other function.
2. High Cohesion 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 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.

Use Case 3:



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 automatically send a message to the waiter that says "hot food," and then the waiter will know that the table's dishes are ready to be served. Finally the waiter will go retrieve the dishes and serve them to the customers.

Design Principles

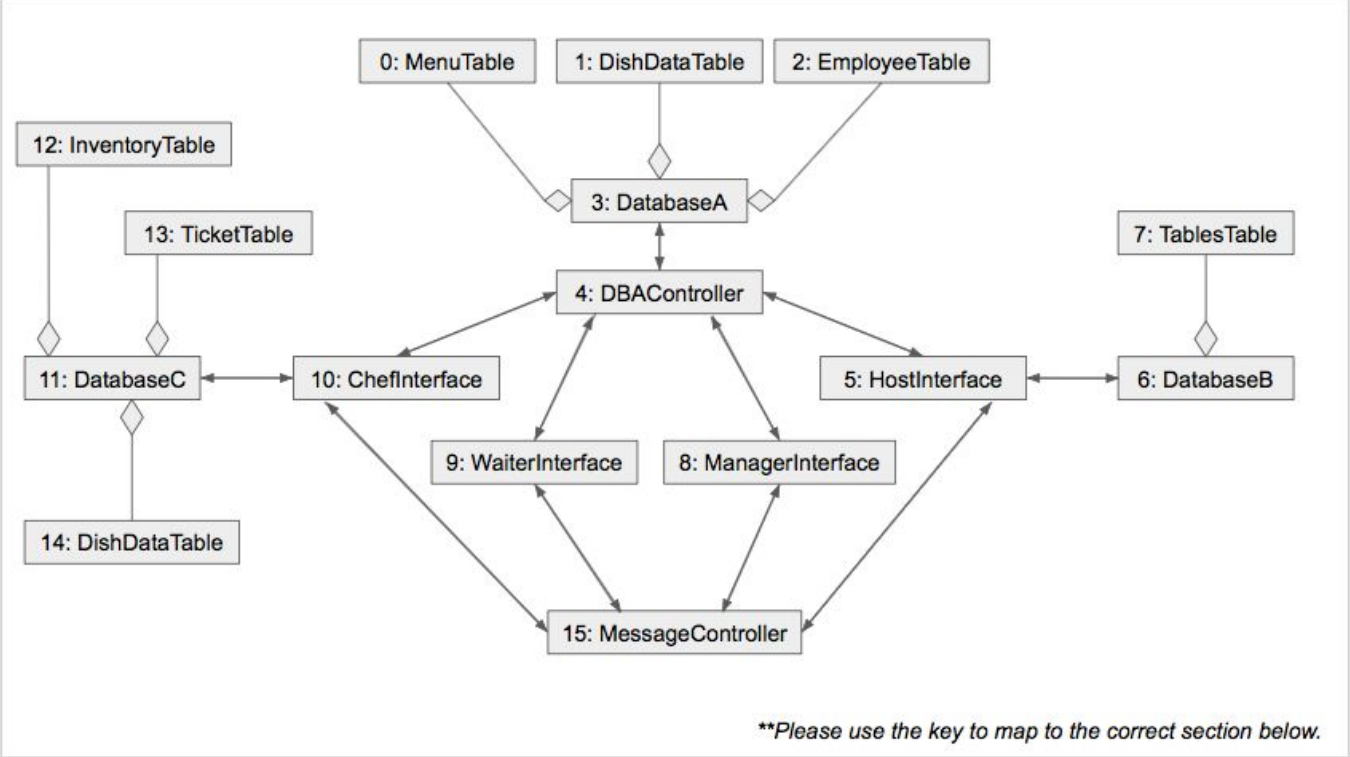
4. The Expert Doer principle is implemented by the:
    a. Message Controller. The message controller is able to decode a message in order it to forward it to the correct destination. Instead of having each device having an additional task of sending a message to a specific device, we gave this task 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.
5. 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.

6. 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.

Class Diagrams & Interface Specification

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/ Owner Interface
    a. Attributes:
        i. HashMap<Long, Employee> listOfEmployees
        ii. HashMap<String, Hashmap<String, DishData>> menu
    b. Functions:
        i. void addEmployee(String Name, char position)
        ii. void removeEmployee(Long id)
        iii. void changeEmployeePosition(Long id, char newPos)
        iv. void addMenuItem(String dishName, String typeOfDish, DishData dd)
        v. void removeMenuItem(String dishName)
        vi. void changePrice(String dishName, Double newPrice)
        vii. char logIn(Long id)
        viii. char logOut(Long id)
        ix. void ownerMessageListener(Message m)
        x. void ownerEventListener(Event e)
        xi. void ownerMessageSender(Message m)
5. Host Interface
8. Manager Interface
    a. Attributes:
        i. ArrayList<Message> listOfMessages
    b. Functions:
        i. void managerMessageListener(Message m)
        ii. void managerEventListener(Event e)
        iii. void managerMessageSender(Message m)
9. Waiter Interface
    a. Attributes:
        i. HashMap<String,HashMap<String,DishData>> menu
        ii. HashMap<Integer,Ticket> listOfOrders
    b. Functions:
        i. void openTicket(Ticket t)
        ii. void printTicket(Ticket t)
        iii. void paid(Ticket t)
        iv. void waiterMessageListener(Message m)
        v. void waiterEventListener(Event e)
        vi. void waiterMessageSender(Message m)
10. Chef Interface
    a. Attributes:
        i. HashMap<String,Ingredient> inventory
        ii. ArrayList<Ticket> ticketQueue
    b. Functions:
        i. void addIngredientToInventory(String ingredientName, Double amountLeft, String unitOfAmount, Double threshold)
        ii. void removeIngredientFromInventory(String ingredientName)
        iii. boolean addTicketToQueue(Ticket t)
        iv. void orderTicketQ()
        v. void changeDish(Dish d, char dishStatus)
        vi. void chefMessageListener(Message m)
        vii. void chefEventListener(Event m)
        viii. void chefMessageSender(Message m)
        ix. void changeTicketColor(Ticket t)
11. Database C
    a. Attributes:
        i. DB databaseC
12. Inventory Table
    a. Attributes:
        i. String ingredientName
        ii. int ingredientID
        iii. double currentAmount
        iv. double threshold
13. Ticket Table

a. Attributes:
   i. HashMap<Integer,Table> listOfReadyTables
   ii. HashMap<Integer,Table> listOfNotReadyTables
b. Functions:
   i. void hostMessageListener(Message m)
   ii. void hostEventListener(Event m)
   iii. void hostMessageSender(Message m)
   iv. void UpdateTable (int tableNumber, char newStatus)
   v. void orderTableList()

6. Database B
   a. Attributes:
      i. DB databaseB
7. TablesTable
   a. Attributes:
      i. int tableNumber
      ii. Date timeLastSat
      iii. char status
      iv. int maxOcc
      v. string waiter

a. Attributes:
   i. int tableNumber
   ii. char ticketStatus
   iii. long waiterID
   iv. date timeStamp
   v. string dishName
   vi. int dishDataID
   vii. char dishStatus

14. Dish Data Table
   a. Attributes:
      i. int dishDataID
      ii. string ingredientName
      iii. double ingredientAmount
      iv. string unit
15. Message Controller
   a. Attributes
      i. ArrayList<Message> listOfPendingMessage
      ii. ArrayList<Ticket> listOfPendingTicket
      iii. HashMap<Char, HashMap<Long, TabAddress>> listOfTabAdresses
   b. Functions
      i. void MessageListener(Message m)
      ii. void messageParser(Message m)
      iii. void sendMessage(Message m, TabAddress ta)
      iv. void sendTicket(Ticket t)
      v. void ticketListener(Ticket t)
      vi. void addTabAddress(Char employeePosition, Long empID, TabletAddr ta)
      vii. void removeTabAddress(Char employeePosition, Long empID)

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.

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.

Data Types & Operation Signatures

Waiter Interface

Attributes

+ menu: HashMap<String, Hashmap<String, Dishdata>>	Hashmap that holds all the items in the menu. Outer map links name of type of dish (Appetizer, Entree, Drinks, Desserts) to a hashmap. This inner hashmap links the name of the dish to its dish data.
+ listOfOrders: HashMap<int,Ticket>	HashMap that holds the tickets for this server. The key is a Double which signifies the table number for that ticket.

Functions

+ void openTicket(Ticket t)	Opens the selected ticket on the screen.
+ void printTicket(Ticket t)	Sends the ticket to the printer to be printed.
+ void paid(Ticket t)	Generates a message to be sent to the host that this ticket's table number has paid.
+ void waiterMessageListener(Message m)	Consistently listens for a message sent by the message controller.
+void waiterEventListener(Event e)	Consistently listens for any external actions by the waiter.
+void waiterMessageSender(Message m)	Sends any necessary messages to the message controller.

Head Chef Interface

Attributes

+ inventory: HashMap<String,Ingredient>	HashMap holding the information about the quantity of ingredients in the inventory.
+ ticketQueue: ArrayList<Ticket>	List of tickets received by the Head Chef, which holds all data for the dishes ordered in the ticket.

Functions

+ void addIngredientToInventory(String ingredientName, Double amountLeft, String unitOfAmount, Double threshold):	Function which allows the chef to add an ingredient to inventory when needed.
+ void removeIngredientFromInventory(String ingredientName) :	Removes ingredient from the inventory when the menu has changed.
+ boolean addTicketToQueue(Ticket t):	Adds a ticket to the Ticket Queue when it has been received by the waiter.
+ void orderTicketQ():	Organizes and color-codes the ticket queue by its respective status and then chronologically.
+void changeDish(Dish d, char dishStatus)	Edits the status of a certain dish when the progress is updated. This calls upon the changeTicketColor function to edit the status of the ticket. Handles the decrementation of the ingredients when a dish has been started as well.
+ void chefMessageListener(Message m)	Consistently listens for a message sent by the message controller.
+ void chefEventListener(Event m)	Consistently listens for any external actions by the chef.
+ void chefMessageSender(Message m)	Sends any necessary messages to the message controller.
+void changeTicketColor(Ticket t)	Edits the status of a certain ticket based on the statuses of the dishes contained in it. Also handles changing the ticket's color to the respective color associated with each status.

Host Interface

Attributes

+ listOfReadyTables: HashMap<Integer,Table>	Map of Tables that are ready to be seated with the key being the table number.
+ listOfNotReadyTables: HashMap<Integer,Table>	Map of Tables that are not ready to be seated with the key being the table number.

Functions

+void hostMessageListener(Message m)	Consistently listens for a message sent by the message controller.
+ void hostEventListener(Event m)	Consistently listens for any external actions by the host.
+ void hostMessageSender(Message m)	Sends any necessary messages to the message controller.
+ void UpdateTable (int tableNumber, char newStatus)	Updates the table's status when seated, paying, not ready, ready.
+ void orderTableList()	Organizes the table queue by its respective status and then chronologically.

Manager Interface

Attributes

+ listOfMessages ArrayList<Message>	List of messages that the Manager has received in chronological order.

Functions

+ void managerMessageListener(Message m)	Consistently listens for a message sent by the message controller.
+void managerEventListener(Event e)	Consistently listens for any external actions by the manager.
+void managerMessageSender(Message m)	Sends any necessary messages to the message controller.

Owner Interface/Database A Controller

Attributes

+ listOfEmployees: HashMap<Long, Employee>	This map maps the employee ID (which is a long) to the employee object.
+ menu: HashMap<String, Hashmap<String, DishData>>	Hashmap that holds all the items in the menu. Outer map links name of type of dish (Appetizer, Entree, Drinks, Desserts) to a hashmap. This inner hashmap links the name of the dish to its dish data.

Functions

+void addEmployee(String Name, char position)	Adds a new employee to the list of employees.
+void removeEmployee(Long id)	Removes an employee from the list of employees using the employee id.
+void changeEmployeePosition(Long id, char newPos)	Updates the position of an employee.
+ void addMenuItem(String dishName, String typeOfDish, DishData dd)	Adds new dish to the menu under the key of typeOfDish (appetizer,entree, or dessert) then under the key of dishName.
+void removeMenuItem(String dishName)	Removes this dish from the menu
+void changePrice(String dishName, Double newPrice)	Changes the price of the dish with the name dishName. The new price will be newPrice.
+char logIn(Long id)	This searches the employee list for the employee id id and returns the position of that employee. It will also generate a message to the message controller giving the message controller the tablet address for that employee.
+char logOut(Long id)	This will remove this employee from the list of employees. It will also generate a message to the message controller notifying that this employee logged out.

+ void ownerMessageListener(Message m)	Consistently listens for a message sent by the message controller.
+void ownerEventListener(Event e)	Consistently listens for any external actions by the owner.
+void ownerMessageSender(Message m)	Sends any necessary messages to the message controller.

Message Controller

Attributes

+ listOfPendingMessage ArrayList<Message>	Holds the list of messages that have not been sent yet.
+ listOfPendingTicket ArrayList<Ticket>	Holds the list of tickets that have not been sent yet.
+ listOfTabAdresses HashMap<Char, HashMap<Long, TabAddress>>	This maps the positions in the restaurant to a map of Tablet addresses. The inner map has the key of the employee id (Long). Each key will correspond to the tablet address being used by that employee.

Functions

+ void messageListener(Message m)	Consistently listens for a message sent by any other systems and adds them to the listOfPendingMessage array.
+void messageParser(Message m)	Decodes the first message in the listOfPendingMessage and calls send Message with the correct parameters.
+ void sendMessage(Message m, TabAddress ta)	Send the message m to the tablet with the address ta.
+ void sendTicket(Ticket t)	Sends the ticket t to the head chef's tablet.
+void ticketListener(Ticket t)	Consistently listens for a ticket sent by any waiter and adds them to the listOfPendingTicket array.

+void addTabAddress(Char employeePosition, Long empID, TabletAddr ta)	Adds an employee's tablet address from the list. This happens when the employee logs in.
+void removeTabAddress(Char employeePosition, Long empID)	Deletes an employee's tablet address from the list. This happens when the employee logs out.

Database A

Attributes

Database:databaseA: DB	Database identifier for Database A
EmployeeTable:employeeID: long	Unique ID of employee
EmployeeTable:name: string	name of the employee
EmployeeTable:position: char	position of the employee
EmployeeTable:tabAddr: string	the IP address of the tablet owned by the employee
MenuTable:dishType: string	Appetizer, entree, or dessert, side, drink
MenuTable:dishName: string	Name of the dish
MenuTable:dishDataID: int	Unique ID for dishData that corresponds to the dishDataID in DishDataTable.
DishDataTable: dishDataID: int	Unique ID for dishData that corresponds to the dishDataID
DishDataTable: ingredientName: string	The name of an ingredient in this dish.
DishDataTable: ingredientAmount: double	The amount of ingredient needed for this dish.
DishDataTable: unit: string	unit of measurement for the ingredient amount

Database B

Attributes

Database:databaseB: DB	Database identifier for Database B

TablesTable:tableNumber: int	The table number
TablesTable: timeLastSat: date	The time the table was last sat
TablesTable:status: char	The status of the table. Can be: ready, oncheck, seated.
TablesTable:maxOcc: int	The maximum amount of people that can be seated at the table
TablesTable:waiter: string	The waiter serving the table

Database C

Attributes

Database:databaseC: DB	Database identifier for Database C
InventoryTable:ingredientName: string	The name of the ingredient
InventoryTable:ingredientID: int	The identifier of the ingredient
InventoryTable:currentAmount: double	The current amount of the ingredient left in the inventory
InventoryTable:threshold: double	The threshold for the ingredient
TicketTable:tableNumber: int	The table number of the ticket
TicketTable:ticketStatus: char	The status of the ticket.
TicketTable: waiterID: long	The identifier of the waiter who send the ticket to the head chef
TicketTable: timeStamp: date	Time that ticket was received.
TicketTable:dishName: string	Name of one dish on this ticket.
TicketTable:dishDataID: int	Unique ID of the dish given by dishName on this ticket.
TicketTable:dishStatus: char	Status of the dish on this ticket.

DishDataTable: dishDataID: int	Unique ID for dishData that corresponds to the dishDataID
DishDataTable: ingredientName: string	The name of an ingredient in this dish.
DishDataTable: ingredientAmount: double	The amount of ingredient needed for this dish.
DishDataTable: unit: string	unit of measurement for the ingredient amount

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	X			X	X	X	X	X	X		
WTS				X					X				X			
WMS				X					X						X	X
MC				X			X	X	X	X					X	X
HEL				X	X	X	X								X	X
Ta				X	X	X										
WEL				X		X			X		X		X		X	X
CEL									X	X	X		X		X	X
TQ											X		X			
TQS										X	X		X			
IU	X	X	X									X		X		
LIC	X	X	X								X	X		X		
MEL								X							X	X
MMS							X	X	X	X					X	X
HMS				X		X	X								X	X
CMS									X	X					X	X
Mes																X

The Traceability Matrix above maps how the Domain Concepts are related to the Software Classes. The Domain Concepts were determined by the Concept Definitions used in the Domain Model. The Software Classes were determined by the Class Descriptions mentioned previously in the report as well. Many Software Classes are used for each Domain Concept, and this illustrates how the software components of our application are interconnected.

System Architecture & System Design

Architectural Styles

The design we use is a client server model with distributed databases. Different clients connect with one integrated server 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.

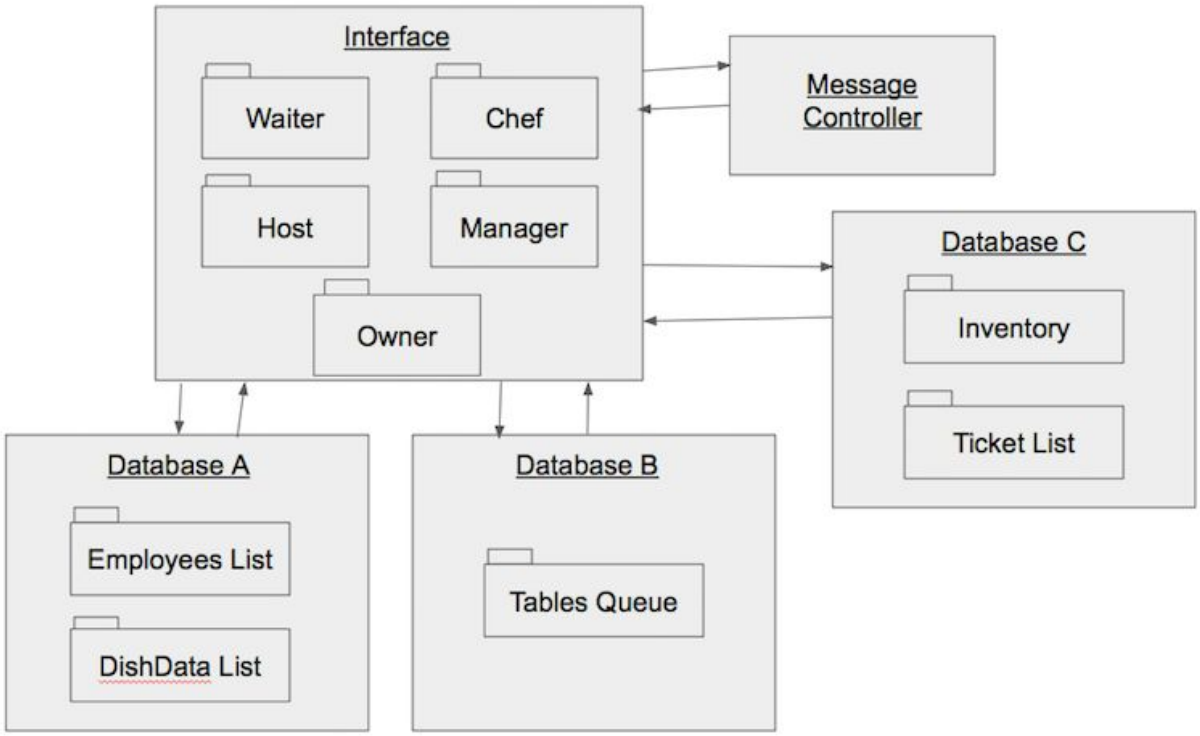
The integrated server holds all of the databases which are used by their respective clients. We used three different databases. The first one holds employee information as well as other static information that can be loaded once a day. This is accessible only to the owner of the restaurant as it contains 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.

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



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, menu items and dish information. 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 ticket list. 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. 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.

Network Protocol

Our 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, *A Service for Servers*, 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.

Global Control Flow

Execution Orderliness

Our application, *A Service for Servers* 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, *A Service for Servers* 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

A Service for Servers 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

Algorithms & Data Structures

Generating Host's Table Queue:

The mathematical model and algorithm for how the host's list of tables is generated that was made in Report 1 has been tweaked. The algorithm is still event driven as events are what cause the change 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 not ready 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:

```
HOST INTERFACE
//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;

//Array list of notifications that the host screen still has to display
ArrayList<Notification> pendingNotifications;

//Constructor:
```

```
public HostInterface(){
    //Code about logging in….
    pendingNotifications = new ArrayList<Notification>();
    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);
    }
}

/*Throughout the day - The host event listener will listen for the host clicking the screen. This will
change the list of tables appropriately.*/
public void hostEventListenter(HostEvent e){
    //if event is a seating a table
    if(e.type == 's'){
            Table t = allTables.get(e.idOfTableNotification);
            if(t==null){
                    return;
            }
            t.seat(e.waiterName);
            readyTables.remove(readyTables.indexOf(e.idOfTableNotification));
            seatedTables.add(e.idOfTableNotification);
    }
    //changing status to ready
    else if(e.type == 'r'){
            Table t = allTables.get(e.idOfTableNotification);
            if(t==null){
                    return;
            }
```

```
            t.setStatus('r');
            paidTables.remove(paidTables.indexOf(e.idOfTableNotification));
            readyTables.add(e.idOfTableNotification);
    }
    else if(e.type == 'n'){//if event is closing a notification
            pendingNotifications.remove(0);
    }
    //if event is sending a notification to manager
    else if(e.type == 'm'){
            hostMessageSender(new Message(new SenderInfo('h'), new RecieverInfo('m'), "Host
Stand needs Assistance."));
    }
    redrawHostScreen(); //update lists that host sees
}

//The host message listener will listen for a paid message from a waiter.
public void hostMessageListener(Message m){
    if(m.getSenderPosition() == 'w'){ //if waiter sent a message
            String content = m.getContent();
            //looks through the content of the message to get the table number
            Integer tableNum = getTableNumberFromMessage(content);
            if(tableNum!=null){
                    Table currTab =allTables.get(tableNum);
                    if(currTab!=null){
                            //if that table is seated
                            if(currTab.getStatus() == 's'){
                                    //change status to paid
                                    currTab.setStatus('p');
                                    seatedTables.remove(seatedTables.indexOf(tableNum));
                                    paidTables.add(tableNum);

                            }
                    }
            }
    }
    else if(m.getSenderPosition()== 'm'){//if manager sent the message =
```

```
        makeNotification(m.getContent());
    }
    redrawHostScreen();
}
```

Generating Total Cost of a Ticket:

This algorithm is used to generate the total cost of a ticket on the waiter's interface. Our algorithm is event driven since the cost 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 inner map maps the dish name to the dish data. The key of the outer hashmap is the type of dish and the data is a hashmap of dishes of that type. The inner hashmap key is the name of the dish and the data associated with the key is the "dish data". The dish data 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 us 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. We are using currDishType in order for the program to know which tab the user is on. This will make looking up the dish data 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 name of that dish as well as the type and so we will be able to use that to look up the dish data for that dish with ease using the menu hashmap.

Pseudo Code:

```
//String currDishType is the name of the tab that the waiter currently has open, this will be used to figure out what type of dish that the waiter is trying to add
// HashMap<String,HashMap<String,DishData>>menu is a hashmap that links  the dish types (appetizer,entrees,desserts) to another map that links dish names to their data.

public void waiterEventListener ( WaiterEvent e ){
    if(e.type =='m' ){
        //notify manager problem with table
    }
    else if(e.type=='b'){
        //go back to previous screen
    }
```

```
    else if(e.type=='p'){
            //send ticket to printer
    }
    else if(e.type=='s'){
            //send ticket to chef
    }
    else if(e.type=='c'){
            //send message to host that you are paid
    }
    else if(e.type =='a' ){ //if you are adding a dish
            double p = getPriceOfDish(e.dishName, currDishType);
                    if(p>=0){
                      currTicket.price = currTicket.price + p;
                      addDishToTicket(e.dishName, currDishType);
                    }
    }
    else if(e.type== 'r'){
            double p = getPriceOfDish(e.dishName, e.dishType);
            if(p>=0){
                      currTicket.price = currTicket.price -p;
                      removeDishFromTicket(e.dishName, e.dishType);
            }
    }
    redrawWaiterScreen();
}
```

Generating Chef's Ticket Queue:

This algorithm generates the queue that the head chef will view on his screen. It is event driven since it is updated when the chef presses a button on his screen to update the status of a dish or when it receives a new ticket from a server. To handle these two separate cases we have a ticket listener which will handle all the tickets being sent in from servers. Then there will be an event listener which will handle all of the button clicks. The Queue that the chef will see will just be one list of tickets. Instead of making one arraylist of tickets we decided to do something similar and make an arraylist for each type of ticket (unstarted, semi started, started, and finished) where the arraylist holds long (which will be the unique ticket id). When the chef draw screen is called this will print out the 4 lists as one list following the order (from top to bottom)

finished, started, semi-started, and unstarted. Having 4 separate lists makes moving a ticket from one status to another (and keeping the correct order) much easier than having one arraylist.

Pseudo Code:

```
//gives you the currTicketNumber you should give the next ticket created
   static long currTicketNumber =0;
//Ticket number to ticket
HashMap<Long, Ticket>ticketLookup;

   //Ticket Queue = holds all ticket orders
 private ArrayList<Long> ticketQueueUnstarted;
 private ArrayList<Long> ticketQueuesemiStarted;
 private ArrayList<Long> ticketQueueStarted;
 private ArrayList<Long> ticketQueueFinished;

//This listener will be used to read incoming tickets from servers and place them on the Q
public void chefTicketListener(Ticket ticket){
   if(ticket!=null){
         //set up the index for this ticket
         ticket.ticketNumber = currTicketNumber;
         //add the ticket to the end of the unstarted list since it is unstarted
         ticketQueueUnstarted.add(currTicketNumber);
         ticketLookup.put(currTicketNumber, ticket);
         currTicketNumber++;
   }
}

//handles all the host actions (like pushing buttons) and updates the screen and list of tables correctly
   public void chefEventListenter(ChefEvent e){
   //if event is changing status of dish
   if(e.type == 's'){
         e.dish.changeStatus(e.newStatusOfDish);
         if(e.newStatusOfDish == 's'){
               decrementInventoryForDish(e.dish);
```

```
            }

            Ticket t = ticketLookup.get(e.ticketNumber);
            char oldstatus = t.updateStatus();
            if( t.getStatus() != oldstatus ){//if the ticket changed its status
                  if(t.getStatus() == 'f'){ //if the ticket is finished
                        //send a message to the server to let them know it is ready
                        chefMessageSender(new Message(new SenderInfo('c'), new
RecieverInfo('s'), "Hot Food."));
                  }
                  changeTicketLocation(oldstatus, t);
            }

      }
      //if event is hitting back button
      else if(e.type == 'b'){
            //move back a screen
      }
      //if event is opening a ticket
      else if(e.type == 'o'){
            //change screen type to show current ticket
            //showTicketOnScreen(e.ticketNumber);
      }
      //if event is pressing get manager button -> sending a notification to manager
      else if(e.type == 'm'){
            chefMessageSender(new Message(new SenderInfo('c'), new RecieverInfo('m'), "Chef
needs Assistance."));
      }
      else if(e.type == 'd'){ //delete a ticket because chef signifies that it was picked up
            ticketQueueFinished.remove(ticketQueueFinished.indexOf(e.ticketNumber));
      }
      redrawChefScreen();

}
```

```
//Changes the list that the ticket is on. ie: takes ticket off of unstarted and adds it to semi started
private void changeTicketLocation(char oldstatus, Ticket t) {
   if(oldstatus == 'u'){
         ticketQueueUnstarted.remove(ticketQueueUnstarted.indexOf(t.ticketNumber));
   }
   else if(oldstatus=='s'){
         ticketQueuesemiStarted.remove(ticketQueuesemiStarted.indexOf(t.ticketNumber));
   }
   else if(oldstatus =='S'){
         ticketQueueStarted.remove(ticketQueueStarted.indexOf(t.ticketNumber));
   }
   else{//finished
         ticketQueueFinished.remove(ticketQueueFinished.indexOf(t.ticketNumber));
   }

   char newStatus = t.getStatus();
   if(newStatus == 'u'){
         ticketQueueUnstarted.add(t.ticketNumber);
   }
   else if(newStatus=='s'){
         ticketQueuesemiStarted.add(t.ticketNumber);
   }
   else if(newStatus =='S'){
         ticketQueueStarted.add(t.ticketNumber);
   }
   else{//finished
         ticketQueueFinished.add(t.ticketNumber);
   }
}
```

Updating Inventory

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 host 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.

User Interface & Implementation

The User Interface for our application is as follows.

Login Screen:

Below are the screens the user will see upon opening the app and logging in.

Design of Tests

Manager:

Test Case Identifier: TC-01 **Function Tested:** managerMessageListener(Message m): void **Pass/Fail Criteria:** Test will pass if manager receives a message.	
Test Procedure	**Expected Results**
Call Function (Pass)	Manager will receive the message.
Call Function (Fail)	Manager does not receive the message.

Test Case Identifier: TC-02

Function Tested: managerEventListener(Event e): void **Pass/Fail Criteria:** Test will pass if the appropriate function is called corresponding to an external action.	
Test Procedure	**Expected Results**
Call Function (Pass)	The appropriate function is called corresponding to an external action.
Call Function (Fail)	The appropriate function is not called corresponding to an external action.

Test Case Identifier: TC-03 **Function Tested:** managerMessageSender(Message m): void **Pass/Fail Criteria:** Test will pass if manager successfully sends a message.	
Test Procedure	**Expected Results**
Call Function (Pass)	Manager successfully sends a message.
Call Function (Fail)	Manager's message does not send.

Chef:

Test Case Identifier: TC-04 **Function Tested:** addIngredientToInventory(String ingredientName, Double amountLeft, String unitOfAmount, Double threshold): void **Pass/Fail Criteria:** Test will pass if the new Ingredient is added to Inventory List.	
Test Procedure	**Expected Results**
Call Function (Pass)	Ingredient is added to Inventory List.
Call Function (Fail)	If similar name already exists, it will generate an error message saying that the ingredient already exists.

Test Case Identifier: TC-05 **Function Tested:** removeIngredientFromInventory(String ingredientName): void **Pass/Fail Criteria:** Test will pass if the ingredient is removed from Inventory List.

Test Procedure	Expected Results
Call Function (Pass)	Ingredient is removed from Inventory List.
Call Function (Fail)	If the ingredient doesn't exist in the inventory, it will generate an error message saying the ingredient doesn't exist.

Test Case Identifier: TC-06
Function Tested: addTicketToQueue(Ticket t): boolean
Pass/Fail Criteria: Test will pass if the Ticket is added to the Chef's Queue.

Test Procedure	Expected Results
Call Function (Pass)	Ticket is added to the Queue when Chef attempts to add it. It will return true.
Call Function (Fail)	Ticket is not added when Chef attempts to add it to the Queue. It will return false.

Test Case Identifier: TC-07
Function Tested: orderTicketQ(): void
Pass/Fail Criteria: Test will pass if the ticket queue is organized and color-coded by each ticket's respective status and then chronologically.

Test Procedure	Expected Results
Call Function (Pass)	The ticket queue is displayed on the head chef's screen in color coded order and then chronological order.
Call Function (Fail)	The ticket queue is displayed on the head chef's screen but not in the correct order.

Test Case Identifier: TC-08
Function Tested: changeDish(Dish d, char dishStatus): void
Pass/Fail Criteria: Test will pass if the status of each dish is correctly updated.

Test Procedure	Expected Results

Call Function (Pass)	The Dish's status will be updated correctly.
Call Function (Fail)	The Dish's status does not update correctly.

Test Case Identifier: TC-09
Function Tested: chefMessageListener(Message m): void
Pass/Fail Criteria: Test will pass if chef receives a message.

Test Procedure	Expected Results
Call Function (Pass)	Chef will receive the message.
Call Function (Fail)	Chef does not receive the message.

Test Case Identifier: TC-10
Function Tested: chefEventListener(Event e): void
Pass/Fail Criteria: Test will pass if the appropriate function is called corresponding to an external action.

Test Procedure	Expected Results
Call Function (Pass)	The appropriate function is called corresponding to an external action.
Call Function (Fail)	The appropriate function is not called corresponding to an external action.

Test Case Identifier: TC-11
Function Tested: chefMessageSender(Message m): void
Pass/Fail Criteria: Test will pass if chef successfully sends a message.

Test Procedure	Expected Results
Call Function (Pass)	Chef successfully sends a message.
Call Function (Fail)	Chef's message does not send.

Test Case Identifier: TC-12

Function Tested: changeTicketColor(Ticket t):void **Pass/Fail Criteria:** Test will pass if the ticket status changes and the color of the ticket changes as the Dish status is updated.	
Test Procedure	**Expected Results**
Call Function (Pass)	Ticket status and color update correctly as each Dish is updated.
Call Function (Fail)	Ticket status and color do not update.

Waiter:

Test Case Identifier: TC-13 **Function Tested:** openTicket(Ticket t): void **Pass/Fail Criteria:** Test will pass if the Waiter opens the selected ticket.	
Test Procedure	**Expected Results**
Call Function (Pass)	Waiter successfully opens selected ticket on the screen.
Call Function (Fail)	Waiter cannot open selected ticket.

Test Case Identifier: TC-14 **Function Tested:** printTicket(Ticket t): void **Pass/Fail Criteria:** Test will pass if the Waiter sends the ticket to the printer.	
Test Procedure	**Expected Results**
Call Function (Pass)	Waiter successfully sends ticket to the printer to be printed.
Call Function (Fail)	Waiter cannot print the ticket.

Test Case Identifier: TC-15 **Function Tested:** paid(Ticket t): void **Pass/Fail Criteria:** Test will pass if a message, which states that a ticket's table number has been paid, is generated.

Test Procedure	Expected Results
Call Function (Pass)	Waiter can generate a message to be sent to the host that the ticket's table number has been paid.
Call Function (Fail)	Waiter cannot generate specified message.

Test Case Identifier: TC-16
Function Tested: waiterMessageListener(Message m): void
Pass/Fail Criteria: Test will pass if the function consistently listens for a message sent by the message controller.

Test Procedure	Expected Results
Call Function (Pass)	Waiter listens for a message sent by Message Controller.
Call Function (Fail)	Waiter doesn't listen for message..

Test Case Identifier: TC-17
Function Tested: waiterEventListener(Event e): void
Pass/Fail Criteria: Test will pass if the function consistently listens for any external actions by the waiter.

Test Procedure	Expected Results
Call Function (Pass)	Function listens for external actions by the Waiter.
Call Function (Fail)	Waiter cannot listen for actions.

Test Case Identifier: TC-18
Function Tested: waiterMessageSender(Message m): void
Pass/Fail Criteria: Test passes if the function sends any necessary messages to the message controller.

Test Procedure	Expected Results
Call Function (Pass)	Function sends necessary messages to the message controller.
Call Function (Fail)	Function doesn't send necessary messages to the message controller.

Host:

Test Case Identifier: TC-19
Function Tested: hostMessageListener(Message m): void
Pass/Fail Criteria: Test will pass if host receives a message.

Test Procedure	Expected Results
Call Function (Pass)	Host will receive the message.
Call Function (Fail)	Host does not receive the message.

Test Case Identifier: TC-20
Function Tested: hostEventListener(Event e): void
Pass/Fail Criteria: Test will pass if the appropriate function is called corresponding to an external action.

Test Procedure	Expected Results
Call Function (Pass)	The appropriate function is called corresponding to an external action.
Call Function (Fail)	The appropriate function is not called corresponding to an external action.

Test Case Identifier: TC-21
Function Tested: hostMessageSender(Message m): void
Pass/Fail Criteria: Test will pass if host successfully sends a message.

Test Procedure	Expected Results
Call Function (Pass)	Host successfully sends a message.
Call Function (Fail)	Host's message does not send.

Test Case Identifier: TC-22
Function Tested: UpdateTable (int tableNumber, char newStatus): void
Pass/Fail Criteria: Test will pass if the table information updates correctly.

Test Procedure	Expected Results
Call Function (Pass)	Table's status updates correctly when seated, paying, not ready, ready.
Call Function (Fail)	Table's status does not update correctly.

Test Case Identifier: TC-23
Function Tested: orderTableList(): void
Pass/Fail Criteria: Test will pass if table queue is ordered by its respective status and then chronologically.

Test Procedure	Expected Results
Call Function (Pass)	Table Queue will be updated correctly and ordered chronologically.
Call Function (Fail)	Table Queue will not be in the correct order or update correctly.

Owner:

Test Case Identifier: TC-24
Function Tested: addEmployee(String Name, char position): void
Pass/Fail Criteria: Test will pass if a new employee is added to the list of employees.

Test Procedure	Expected Results
Call Function (Pass)	A new employee is added to the list of employees.
Call Function (Fail)	If an employee with the same name and position already exists in the list of employees, then the new employee isn't added to the list.

Test Case Identifier: TC-25
Function Tested: removeEmployee(Long id): void
Pass/Fail Criteria: Test will pass if the employee is removed from the list of employees .

Test Procedure	Expected Results
Call Function (Pass)	The employee is removed from the list of employees.
Call Function (Fail)	If the employee doesn't exist in the list of employees, then it would return an error message stating that the employee doesn't exist.

Test Case Identifier: TC-26
Function Tested: changeEmployeePosition(Long id, char newPos): void
Pass/Fail Criteria: Test will pass if the employee's position is correctly updated .

Test Procedure	Expected Results
Call Function (Pass)	The employee's position is correctly updated.
Call Function (Fail)	If the employee id doesn't exist, it would return an error message stating that the employee doesn't exist.

Test Case Identifier: TC-27
Function Tested: addMenuItem(String dishName, String typeOfDish, DishData dd): void
Pass/Fail Criteria: Test will pass if the new menu item is added to the menu .

Test Procedure	Expected Results
Call Function (Pass)	The new menu item is added to the menu.
Call Function (Fail)	If a menu item with the same dishName and typeOfDish exists in the menu, it would return an error message stating that this menu item exists.

Test Case Identifier: TC-28
Function Tested: removeMenuItem(String dishName): void
Pass/Fail Criteria: Test will pass if the menu item is removed from the menu.

Test Procedure	Expected Results
Call Function (Pass)	The menu item is removed from the menu.
Call Function (Fail)	If the menu item doesn't exist in the menu, it would return an error message stating that the menu item doesn't exist.

Test Case Identifier: TC-29
Function Tested:changePrice(String dishName, Double newPrice): void
Pass/Fail Criteria: Test will pass if the price of the dish changes to the new price .

Test Procedure	Expected Results
Call Function (Pass)	The price of the dish changes to the new price.
Call Function (Fail)	If the dish with dishName doesn't exist, it would return an error message stating that this dish doesn't exist.

Test Case Identifier: TC-30
Function Tested: logIn(Long id): char
Pass/Fail Criteria: Test will pass if the employee id is found in the list of employees and the position of the employee is returned, logging the employee in .

Test Procedure	Expected Results
Call Function (Pass)	The employee id is found in the list of employees and the position of the employee is returned.
Call Function (Fail)	If the employee id is not found in the list of employees, zero is returned representing no employee.

Test Case Identifier: TC-31
Function Tested: logOut(Long id): char
Pass/Fail Criteria: Test will pass if the employee id is found from the list of the employees and logs the employee out of the system.

Test Procedure	Expected Results
Call Function (Pass)	The employee id is found from the list of the employees and logs the employee out of the system. It would return the position of that employee.
Call Function (Fail)	If the employee id doesn't exist in the list of employees, zero is returned representing no employee and the employee can't be logged out.

Message Controller:

Test Case Identifier: TC-32
Function Tested: messageListener(Message m): void
Pass/Fail Criteria: Test will pass if the Message Controller consistently listens for a message sent by any other systems and adds them to the listOfPendingMessage array.

Test Procedure	Expected Results
Call Function (Pass)	Message Controller listens for message and adds them to listOfPendingMessage array.
Call Function (Fail)	Message Controller cannot listen for message or add them to listOfPendingMessage array.

Test Case Identifier: TC-33
Function Tested: messageParser(Message m): void
Pass/Fail Criteria: Test will pass if the Message Controller reads in the message and calls sendMessage with the correct parameters.

Test Procedure	Expected Results
Call Function (Pass)	Message Controller successfully parses message and calls sendMessage.
Call Function (Fail)	Message Controller cannot parse message or call sendMessage.

Test Case Identifier: TC-34
Function Tested: sendMessage(Message m, TabAddress ta): void
Pass/Fail Criteria: Test will pass if Message Controller sends a given Message m to the the tablet with address (TabAddress) ta.

Test Procedure	Expected Results
Call Function (Pass)	Message Controller successfully sends a given message to tablet with a specified address.
Call Function (Fail)	Message Controller cannot send a given message to tablet with a specified address.

Test Case Identifier: TC-35
Function Tested: sendTicket(Ticket t): void
Pass/Fail Criteria: Test will pass if Message Controller sends Ticket t to the Head Chef's tablet.

Test Procedure	Expected Results
Call Function (Pass)	Message Controller successfully sends ticket to Head Chef's tablet.
Call Function (Fail)	Message Controller cannot send ticket to Head Chef's tablet.

Test Case Identifier: TC-36
Function Tested: ticketListener(Ticket t): void
Pass/Fail Criteria: Test will pass if the Message Controller consistently listens for a ticket

sent by any waiter and adds them to the listOfPendingTicket array.	
Test Procedure	**Expected Results**
Call Function (Pass)	Message Controller listens for ticket from any waiter and add them to the listOfPendingTicket array.
Call Function (Fail)	Message Controller cannot listen for ticket from any waiter and cannot add them to the listOfPendingTicket array.

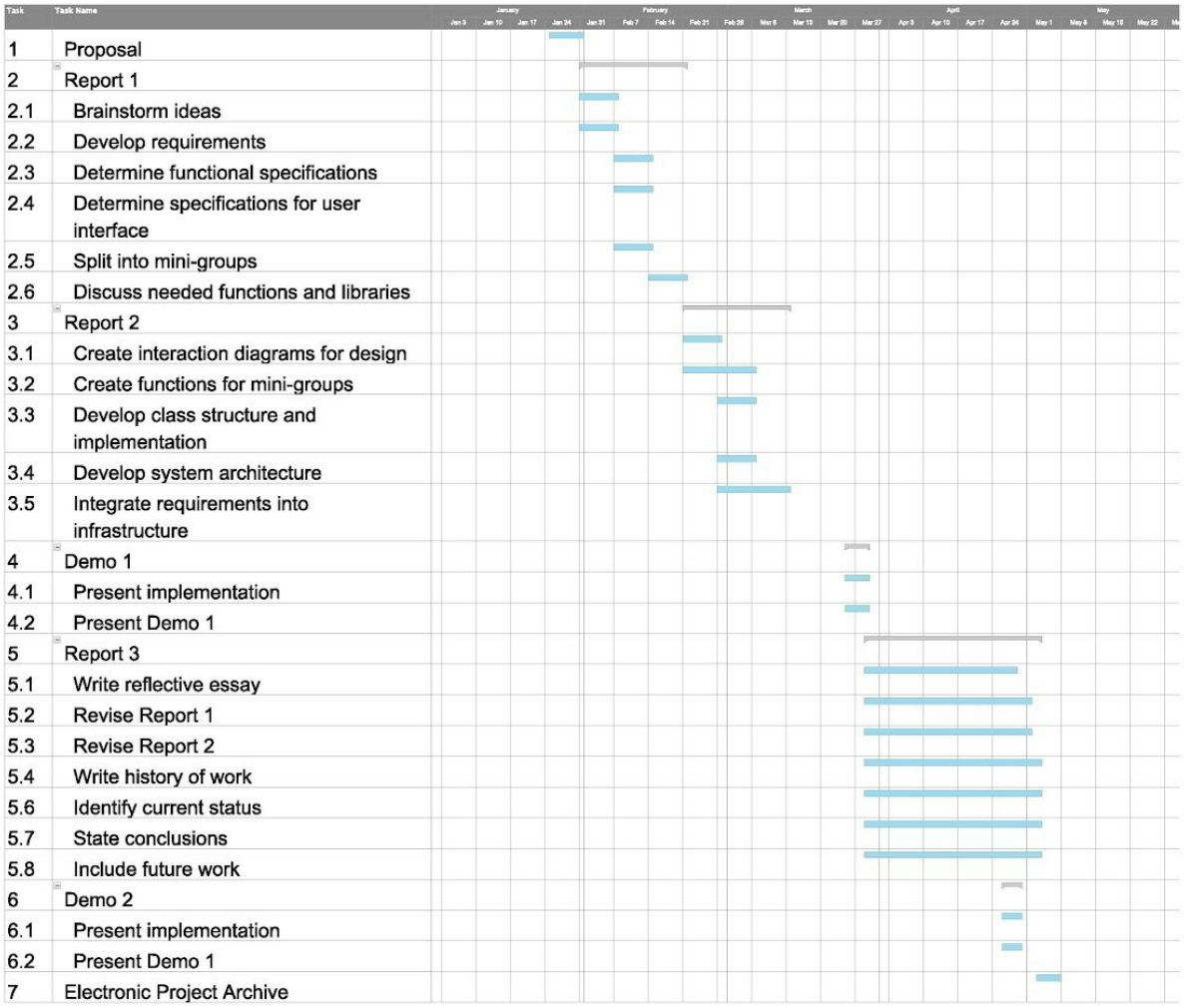
Test Case Identifier: TC-37 **Function Tested:** addTabAddress(Char employeePosition, Long empID, TabletAddr ta): void **Pass/Fail Criteria:** Test will pass if a Message Controller adds an employee's tablet address from the list. This happens when the employee logs in.	
Test Procedure	**Expected Results**
Call Function (Pass)	Message Controller adds an employee's tablet address from the list.
Call Function (Fail)	If the employee doesn't exist in the list of employees, it would return an error message stating that the employee doesn't exist.

Test Case Identifier: TC-38 **Function Tested:** removeTabAddress(Char employeePosition, Long empID): void **Pass/Fail Criteria:** Test will pass if the Message Controller deletes an employee's tablet address from the list. This happens when the employee logs out.	
Test Procedure	**Expected Results**
Call Function (Pass)	The Message Controller deletes an employee's tablet address from the list. This happens when the employee logs out.
Call Function (Fail)	If the employee doesn't exist in the list of employees, it would return an error message stating that the employee doesn't exist.

Project Management & History of Work

a. Project Management

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.



b. 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 a few times 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.

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.

Current Status

The team is currently in the process of developing the Graphical User Interface (GUI) and database for our application. We are working on different portions of the project simultaneously. We just completed Part 2 of Report 2, and Part 3 for completion of Report 2 is

currently underway.

Future Status

Mindful of the approaching deadline for our First Demo, we will be working to complete the development of our application, in terms of constructing the GUI and database. We are also looking to get ahead on our project by having each individual member's work portion done prior to meeting up in person so that we can effectively utilize time spent compiling the report as a group.

c. Product Ownership

We decided to break the larger group up into mini groups of 2 people to improve the efficiency of our group. We chose the mini groups based on team familiarity and skill balance. We still worked as a whole group for certain parts of the project.

d. Breakdown of Responsibilities

Team Head Chef:

Mini-Group Members: Annie Antony and Nishtha Sharma

Completed Tasks:

- Understand how to use GitHub.
- Plan out the foundation of all the Head Chef classes.

Current Tasks:

- Create functions to organize the order of the tickets received by our color coordination set up.
- Create foundation of all remaining head chef classes.
- Create an event listener for the tickets received from the waiter and use the above function to sort them.

Future Tasks:

- Make the Head Chef GUI and link the functions to the it.
- Understand how to use SQL to create a database.
- Understand how to get data from SQL and use it in our functions.
- Create the Database associated with the Head Chef and link to the interface.

Team Waiter:

Mini-Group Members: Christina Segerholm and Athira Haridas

Completed Tasks:

- Understand how Waiter is going to communicate with head chef, host, and manager.
- Started coding classes that waiter will use:
  - Dish, DishData, Ticket, Table, waiterInterface
- Found a way to get the whole menu from Database A into the Waiter Interface when a Waiter logs on.
- Researched how to make waiter GUI using java swing library.
- Create functions for sending messages to message controller

Current Tasks

- Continue developing classes related to waiter interface
- Create Waiter GUI

Future Tasks:

- Make event listener for waiter which should listen for any buttons on the screen pressed or any messages being sent to it.
- Start to determine shared interface

Team Host:

Mini-Group Members: Emma Roussos and Christina Parry

Completed Tasks:

- Created a first draft for what the host interface will look like.
- Understood the interactions between the Host and database.
- Understood how Domain Concepts relate to Software Classes.
- Understood attributes and functions needed for Host.
- Created coding environment.

Current Tasks:

- Research how to create a GUI for the host interface using Java Swing.
- Implement Host Interface.
- Implement functions for Host.

Future Tasks:

- Understand integration of host functions with database.
- Debug Host implementations

Whole Team:

Completed Tasks: Created an outline (initial draft) of the overall system.

Current Tasks:

- Develop a complete understanding of the interactions between the different subsystems in the software application.

- Develop the shared infrastructure library piece by piece.
- Research how to create a GUI for the manager interface.
- Understand how to use SQL server to get databases up and running.

Future Tasks:

- Develop Database A controller to manage data flow into and out of databases.
- Create a message controller to allow communication between all subsystems.
- Create the manager user interface.

References

Previous Group Projects:

- Group 3 Spring 2015

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