14:332:452:01 Software Engineering Spring ‘14
Group 4
Full Report 2, Part 1&2&3 Submission: March 19th ,2014

WHY WAIT

A Restaurant Automation System

http://mitulgada.wix.com/whywait

Group Members:
Amgad Armanus
Jake Chou
Mitul Gada
Avni Patel
Nirjan Thayaparan
Diego Urquiza
Christian Youssef

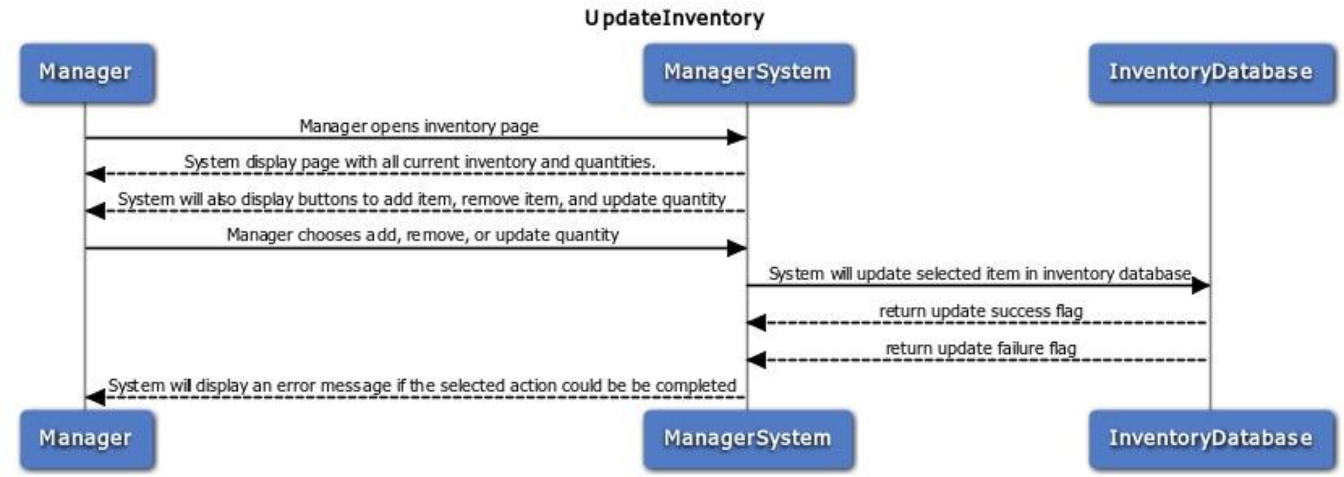
Project Management

All team members contributed equally.

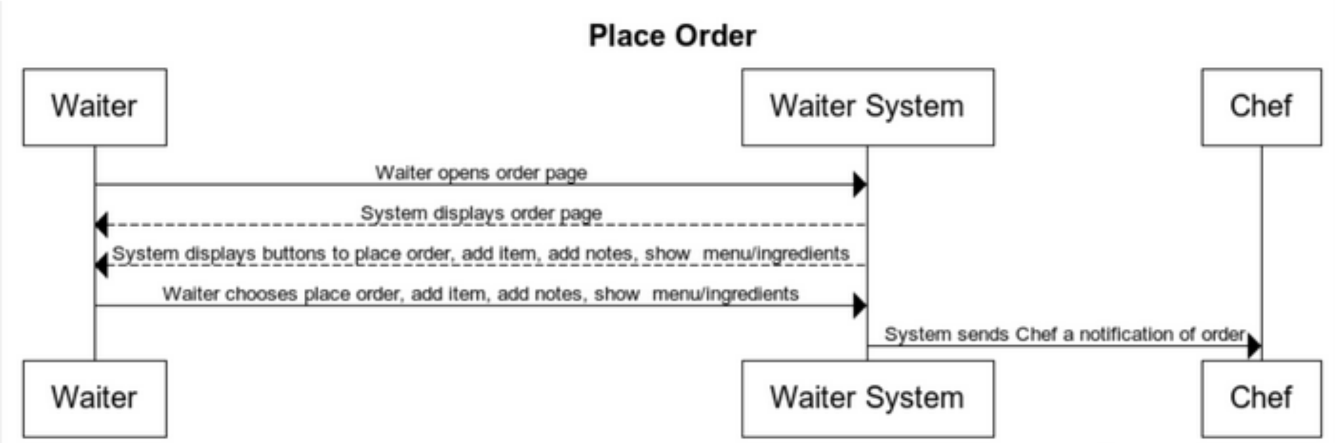
Table of Contents

1). Interaction Diagrams

1. Interaction Diagrams

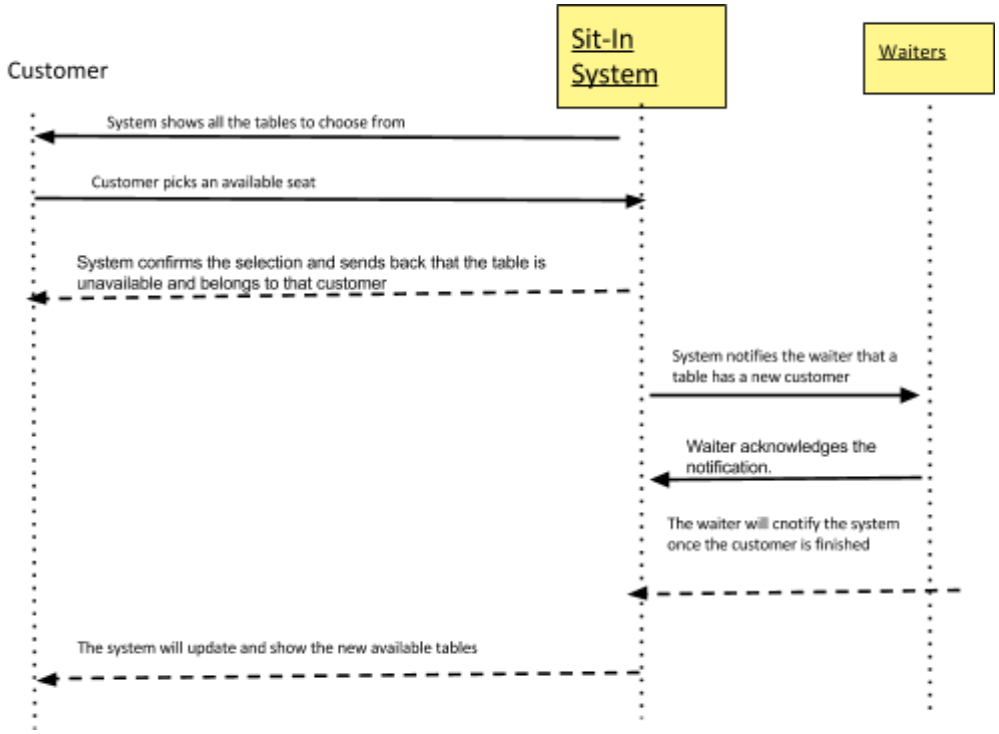

UpdateInventory
Manager
ManagerSystem
InventoryDatabase
Manager opens inventory page
System display page with all current inventory and quantities.
System will also display buttons to add item, remove item, and update quantity
Manager chooses add, remove, or update quantity
System will update selected item in inventory database
return update success flag
return update failure flag
System will display an error message if the selected action could be be completed
Manager
ManagerSystem
InventoryDatabase

The ManagerSystem interacts with the inventory database and displays the results to the manager screen. In case of a failure to update the database, the ManagerSystem will display an error message to inform the manager.

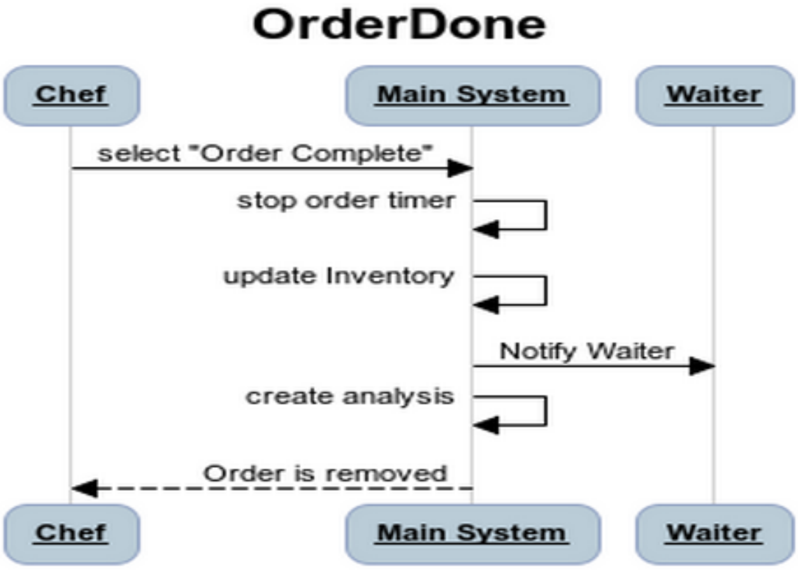

Place Order
Waiter
Waiter System
Chef
Waiter opens order page
System displays order page
System displays buttons to place order, add item, add notes, show menu/ingredients
Waiter chooses place order, add item, add notes, show menu/ingredients
System sends Chef a notification of order
Waiter
Waiter System
Chef

The Waiter System interacts with and sends a notification the Chef's PC. The Waiter Screen allows the Waiter to place an order, add items, add notes, and see the menu/ingredients.

UpdateTable:

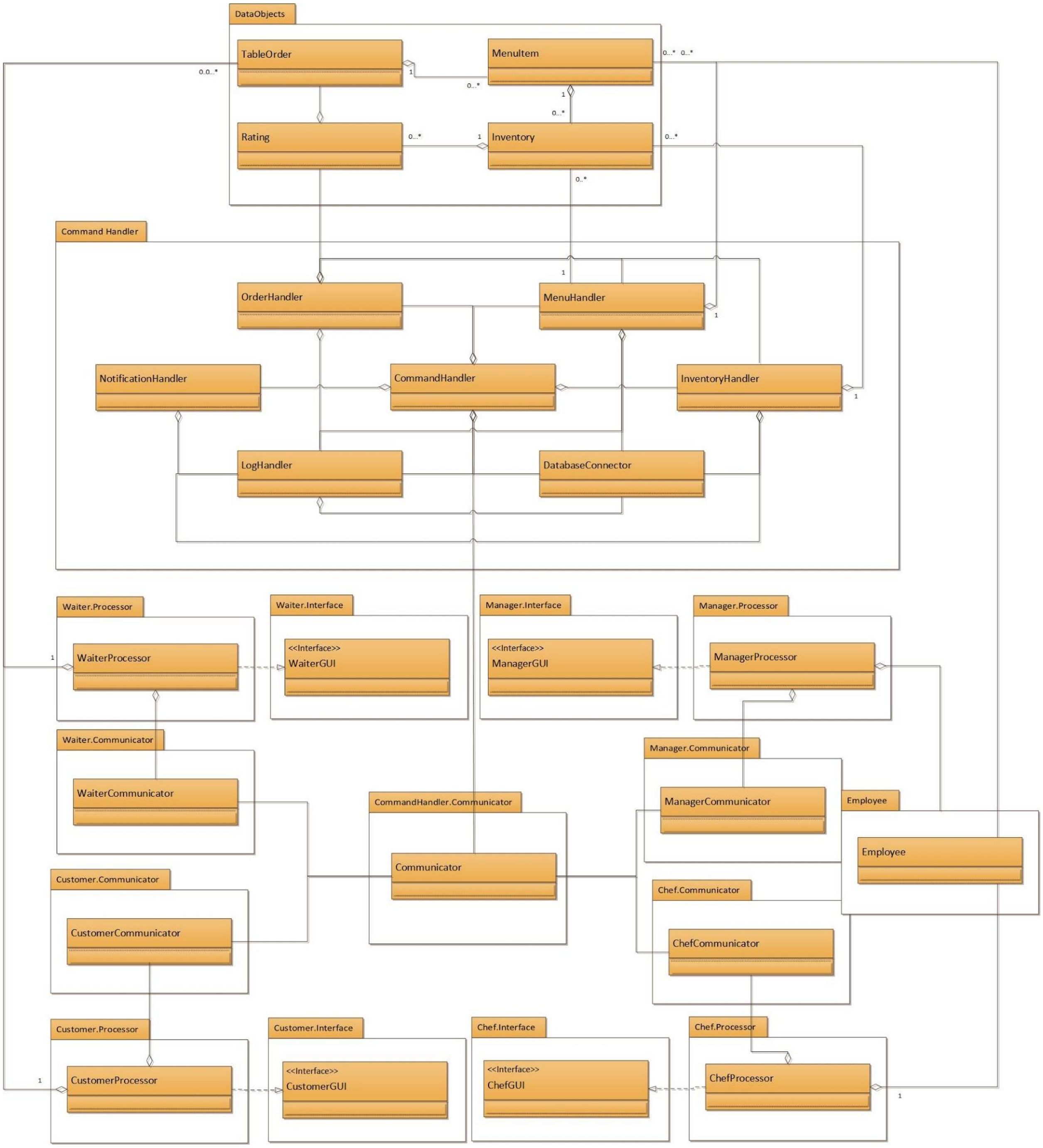

Customer
Sit-In System
Waiters
System shows all the tables to choose from
Customer picks an available seat
System confirms the selection and sends back that the table is unavailable and belongs to that customer
System notifies the waiter that a table has a new customer
Waiter acknowledges the notification.
The waiter will cnotify the system once the customer is finished
The system will update and show the new available tables

The customer sign in PC will be used by both the customers and the waiters. The customers will see the status of all the tables in real time and be able to choose the available ones. The system will then change it so that the next customers will see that the table is unavailable. Once finished, the waiter will update the system so it can send back to the customer PC when an unavailable table will be ready.



The Chef PC will utilize this sequence in order to send a notification through the system to the waiter that an order has been completed and is ready to be served to the customer. The chef will press the "Order Complete" option, which will tell the system to stop the order times for that specific order, update the inventory in response to the order completed, notify the waiter that the meal is ready to be served, and create an analysis based on the order. Through this option, efficiency throughout will be increased because of all the actions that happen at once through the system instead of manually and one by one.

2. Class Diagram and Interface Specification

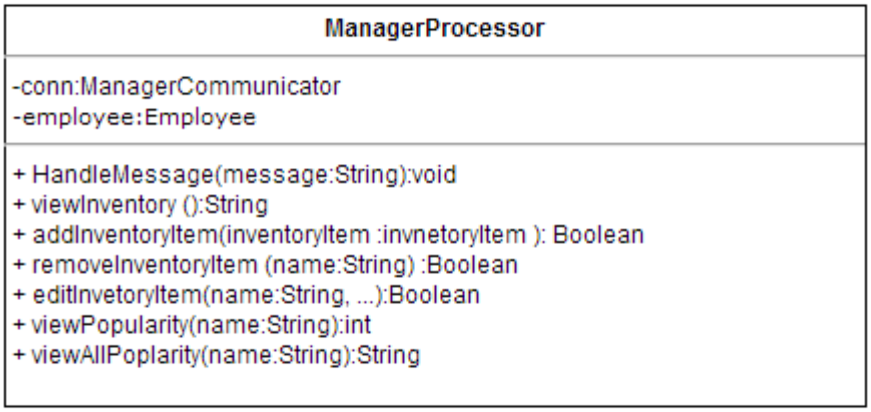


Data Types and Operation Signatures

Note For all of our classes we are going to be using spring 2013 ‘Auto-Serve’ as reference for their classes and we will be optimizing and adding newer functions.

ManagerProcessor

ManagerProcessor is responsible for handling the requests given by the CustomerGUI.



ManagerProcessor

-conn:ManagerCommunicator
-employee:Employee

+ HandleMessage(message:String):void
+ viewInventory ():String
+ addInventoryItem(inventoryItem :invnetoryItem): Boolean
+ removeInventoryItem (name:String) :Boolean
+ editInvetoryItem(name:String, ...):Boolean
+ viewPopularity(name:String):int
+ viewAllPoplarity(name:String):String

Attributes

-conn:ManagerCommunicator
-employee:Employee

Methods

+HandleMessage(message:String):void

Used to handle requests that are being passed to this function.

+viewInventory():String

Views inventory items that are in the system.

+addInventoryItem(inventoryItem:Inventory):Boolean

Allows the user to add an inventory item to the system.

+editInventory(name:String,…):Boolean

Allows used to edit a selected inventory item.

+viewPopularity(name:String):int

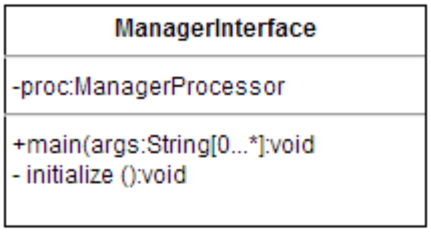
Displays the popularity of the selected menu item.

+viewAllPopularity(name:String):String

Displays popularity of all items on the menu.

ManagerInterface

ManagerGUI is the what the manager interacts with on their display. This is responsible for communication between the manager and the system.



ManagerInterface

-proc:ManagerProcessor

+main(args:String[0...*]:void
- initialize ():void

Attributes

-proc:ManagerProcessor

Processes all requests made and calls the appropriate function.

Methods

+main(args:String[0….*]):void

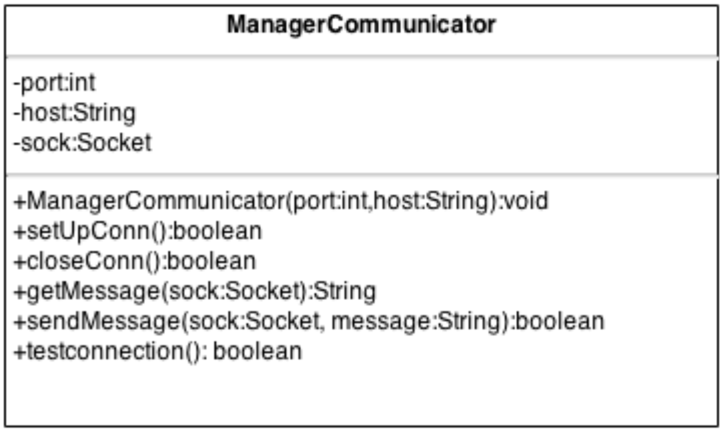
The main function that calls the initialize to prompt the initialization of the GUI.

-initialize():void

Creates the GUI that is displayed for the manager.

ManagerCommunicator

The ManagerCommunicator class will take care of all the connections to the server from the client side. The functions in this class will be responsible for sending requests to the CommandHandler as well as sending information to the ManagerGUI.



ManagerCommunicator

-port:int
-host:String
-sock:Socket

+ManagerCommunicator(port:int,host:String):void
+setUpConn():boolean
+closeConn():boolean
+getMessage(sock:Socket):String
+sendMessage(sock:Socket, message:String):boolean
+testconnection(): boolean

Attributes

-port:int

The port number of the client.

-host:String

protocol string that will be string compared in order to make sure it has the name pipe sql protocol

-sock:Socket

The socket of the client used to communicate to the server

Methods

+ManagerCommunicator(port:int,host:String):void

ManagerCommunicator will set up the port number and the host string

+setUpConn():boolean

This function will use port number and string initialized from MangerCommunicator in order to set up the socket connection

+closeConn():boolean

Terminates the socket connection

+getMessage(sock:Socket):String

Gets request coming into the client socket

+sendMessage(sock:Socket,message:String):boolean

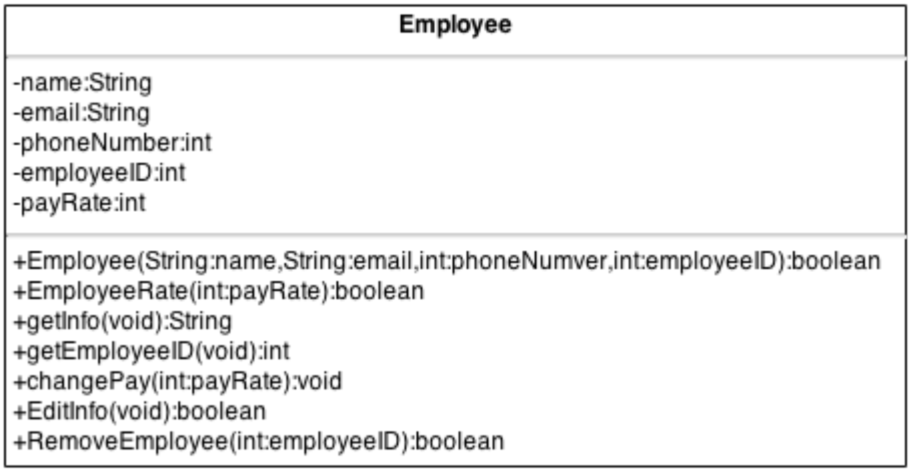
Takes the socket information and sends it as a string which is formated to sql protocols to the server alongside with a message.

+testConnection():boolean

Test the connection to the server. If the value returns false as in failed connection, it will invoke setUpConn

Employee

The Employee class will set up all or remove employees from the system. This class will be friends with the ManagerProcessor class since we want the manager to have the ability to change employee information on the go.



Employee

-name:String
-email:String
-phoneNumber:int
-employeeID:int
-payRate:int

+Employee(String:name,String:email,int:phoneNumver,int:employeeID):boolean
+EmployeeRate(int:payRate):boolean
+getInfo(void):String
+getEmployeeID(void):int
+changePay(int:payRate):void
+EditInfo(void):boolean
+RemoveEmployee(int:employeeID):boolean

Attributes

-name:String

First and last name of the employee.

-email:String

Employee primary email information.

-phoneNumber:int

Employee primary contact phone number.

-employeeID:int

ID that will be used in order to keep better track of employees.

-payRate:int

Hourly rate of employees

Methods

+Employee(String:name,String:email,int:phoneNumber,int:employeeID):boolean

This function will set up a new employee into the system based off of there name,email, and phone number. After successfully entering a valid new employee the system will display employee information as will as generated employee ID.

+EmployeeRate(int:payRate,int:employeeID):boolean

Manager can set initial pay rate for the employee.

+getInfo(int:employeeID):String

Pull up all of employee’s information.

+getEmployeeID(void):int

Bring up a list of all current employees organized by ID order.

+changePay(int:payRate,int:employeeID):void

Change the hourly rate of employees payroll.

+EditInfo(int:employeeID):boolean

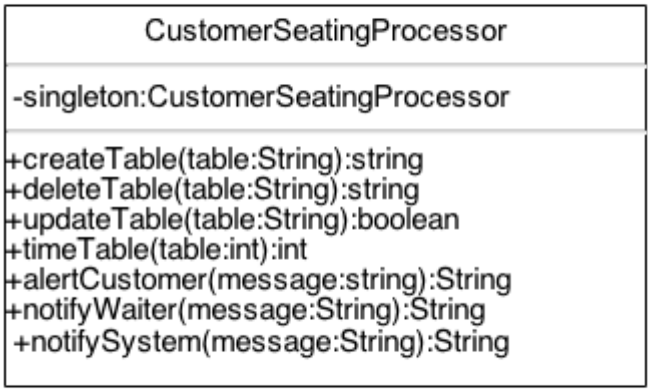
Allows manager to edit basic employee information (name,email, or phonenumber).

+RemoveEmployee(int:employeeID):boolean

Removes employee completely from the system (except payroll information).

CustomerSeatingProcessor

The Customer Seating Processor's task organizes all of the tables and gives the status of each one to the customer and waiters. The customers will be able to see all of the tables and the status of each and choose the available ones. These requests will be sent to the waiters to notify them to bring the customer to the selected table. The system will then update for the next customer.



CustomerSeatingProcessor

-singleton:CustomerSeatingProcessor

+createTable(table:String):string
+deleteTable(table:String):string
+updateTable(table:String):boolean
+timeTable(table:int):int
+alertCustomer(message:string):String
+notifyWaiter(message:String):String
+notifySystem(message:String):String

Attributes

-singleton:CustomerSeatingProcessor

The function only calls to itself

Methods

+createTable(table:String):string

This method will create a new string and insert a new table.

+deleteTable(table:String):string

This method will delete an existing string and delete a table

+updateTable(table:String):boolean

This method will change the status of a table. There are only 2 statuses, available or unavailable.

+timeTable(table:int):int

This method will show the current time of the table of how long the customer was at that table to show the customers and waiters and estimate time of how much longer they will take.

+alertCustomer(message:string):String

This alert will show the customer when a table has become available again if a customer at the table has just left or change the table they just selected to unavailable for the next customer to see.

+notifyWaiter(message:String):String

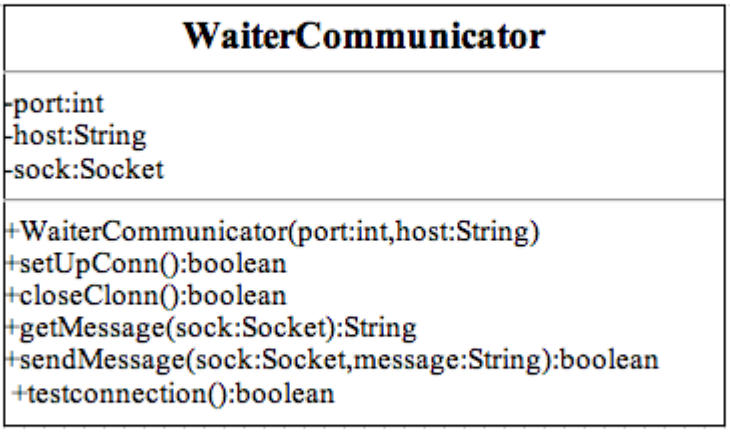
This alert will notify the waiter that a customer has just chosen a table and is ready to be seated.

+notifySystem(message:String):String

The system will update accordingly to give real time updates and statuses of every table

Waiter.Communicator

The WaiterCommunicator is responsible for all communications between the CommandHandler and the WaiterGUI. This class is actively listening for requests from the CommandHandler and is responsible for all changes made to the WaiterGUI.



WaiterCommunicator

-port:int
-host:String
-sock:Socket

+WaiterCommunicator(port:int,host:String)
+setUpConn():boolean
+closeClonn():boolean
+getMessage(sock:Socket):String
+sendMessage(sock:Socket,message:String):boolean
+testconnection():boolean

Attributes

-port:int

The port through which the class listens through.

-host:String

The Hostname of the local computer

-sock:Socket

The socket that sends and receives requests on.

Methods

+WaiterCommunicator(port:int,host:String)

The constructor.

+setUpConn():boolean

The method used to setup the connection for the socket.

+closeConn():Boolean

The method used to close the connection on the socket.

+getMessage(sock:Socket):String

The method used to receive a request on a connected socket

+sendMessage(sock:Scoket,message:String):Boolean

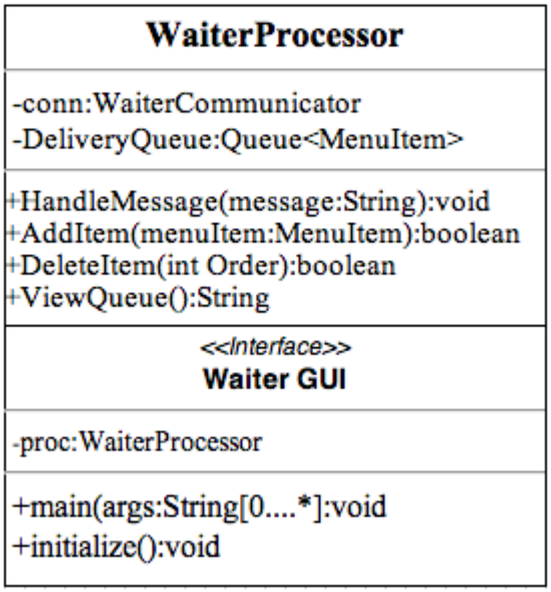
The method used to send a message from the socket.

+testConn():Socket

The method used to test the socket’s connection.

Waiter.Processor

The WaiterProcessor is responsible for the maintaining the OrderQueue locally while also handing the requests given by the WaiterGUI.


WaiterProcessor
-conn:WaiterCommunicator
-DeliveryQueue:Queue<MenuItem>
+HandleMessage(message:String):void
+AddItem(menuItem:MenuItem):boolean
+DeleteItem(int Order):boolean
+ViewQueue():String
<<Interface>>
Waiter GUI
-proc:WaiterProcessor
+main(args:String[0....*]:void
+initialize():void

Attributes

-conn:WaiterCommunicator

This is the socket connection used to communicate with other components in the system.

-DeliveryQueue:Queue<MenuItem>

This is the container for the queue that holds the items that were ordered and are ready to be delivered to customers.

Methods

+HandleMessage(message:String):void

This method handles all the messages the waiter sends to the Chef.

+AddItem(int Order):Boolean

This method adds an order to the Deliveryqueue and gives an order number.

+DeleteItem(int Order):Boolean

This method removes an order form the Deliveryqueue based on the order number.

+ViewQueue():ArrayList<MenuItem>

This methods returns the current delivery queue.

WaiterGUI

The WaiterGUI is the front end of the system. It is the interface that the Waiter uses and is the liaison between the Waiter and the system. This class will be using the WaiterProcessor to aid in processing the requests by the Waiter.

Attributes

-proc:WaiterProcessor

This object is used to process all the requests.

Methods

+main(args:String[0….*]):void

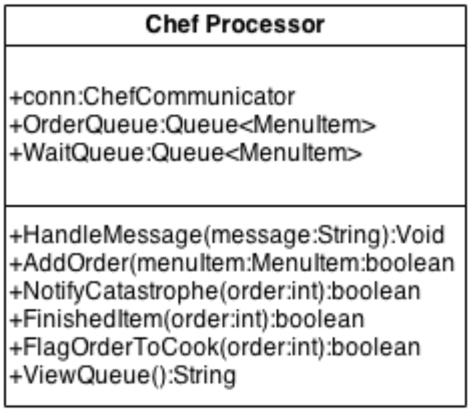
This method is used to initialize the GUI.

-intialize():void

This method creates the GUI

Chef.Processor

The ChefProcessor is responsible for the maintaining the OrderQueue and BeingCookedQueue locally while also handling the requests given by the chefGUI. An example of a request is: Flagging order done which has to send the order to the WaiterGUI.



Chef Processor

+conn:ChefCommunicator
+OrderQueue:Queue<MenuItem>
+WaitQueue:Queue<MenuItem>

+HandleMessage(message:String):Void
+AddOrder(menuItem:MenuItem:boolean
+NotifyCatastrophe(order:int):boolean
+FinishedItem(order:int):boolean
+FlagOrderToCook(order:int):boolean
+ViewQueue():String

Attributes

+conn:ChefCommunicator

This object is used to send and receive requests.

+OrderQueue:Queue<MenuItem>

This object holds the menuItems on the OrderQueue

+WaitQueue:Queue<MenuItem>

This object holds the menuItem on the Orders to cook.

Methods

+HandleMessage(message:String):void

This method handles any message passed to the chef. 93

+AddOrder(menuItem:MenuItem):boolean

This method add a menu item to the chef's ready queue .

+DeleteItem(order:int):boolean

This method removes an item from the chef's ready queue and puts it on the waiter queue.

+NotifyCatastrophe(order:int):Boolean

This method will notify the controller of a catastrophe and to halt the current queue.

+FinishedItem(order:int):Boolean

This method will take an item from the wait queue and flag it as done. This will send a message to the controller to forward the item to the waiter to be delivered.

+FlagOrderToCook(order:int):Boolean

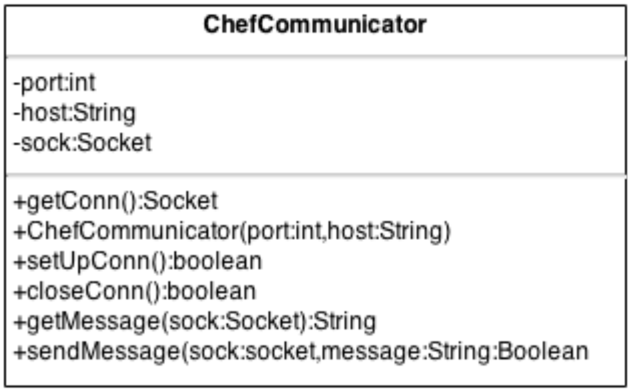
This method will flag an order to be cooked which will move it to the wait queue.

+ViewQueue():String

This method will return the current queue for the chef.

Chef.Communicator

The ChefCommunicator is responsible for sending and receiving any communication between the ChefGUI and the CommandHandler. This class is actively listening for requests from the CommandHandler and can be responsible for any changes made in the ChefGUI.



ChefCommunicator

-port:int
-host:String
-sock:Socket

+getConn():Socket
+ChefCommunicator(port:int,host:String)
+setUpConn():boolean
+closeConn():boolean
+getMessage(sock:Socket):String
+sendMessage(sock:socket,message:String:Boolean

Attributes

-port:int

The port through which the class is going to listen through.

-host:String

The Hostname of the local computer

-sock:Socket

The socket that going to be used to send and receive requests on.

Methods

+ChefCommunicator(port:int,host:String)

The constructor used to initialize. 94

+setUpConn():boolean

The method used to setup the connection on the socket.

+getConn():Socket

The method used to listen and return any incoming information

+closeConn():Boolean

The method used to close the connection on the socket.

+getMessage(sock:Socket):String

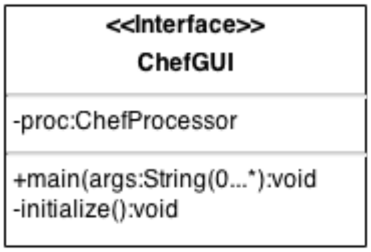
The method used to receive a request on a connected socket

+sendMessage(sock:Scoket,message:String):Boolean

The method used to send a message on the socket.

Chef.Interface

The ChefGUI is the front end of the system and is responsible for the interface between the Chef and the system. This class will be using the ChefProcessor to aid in processing the requests by the chef.



Attributes

-proc:ChefProcessor

This object is used to process all the requests.

Methods

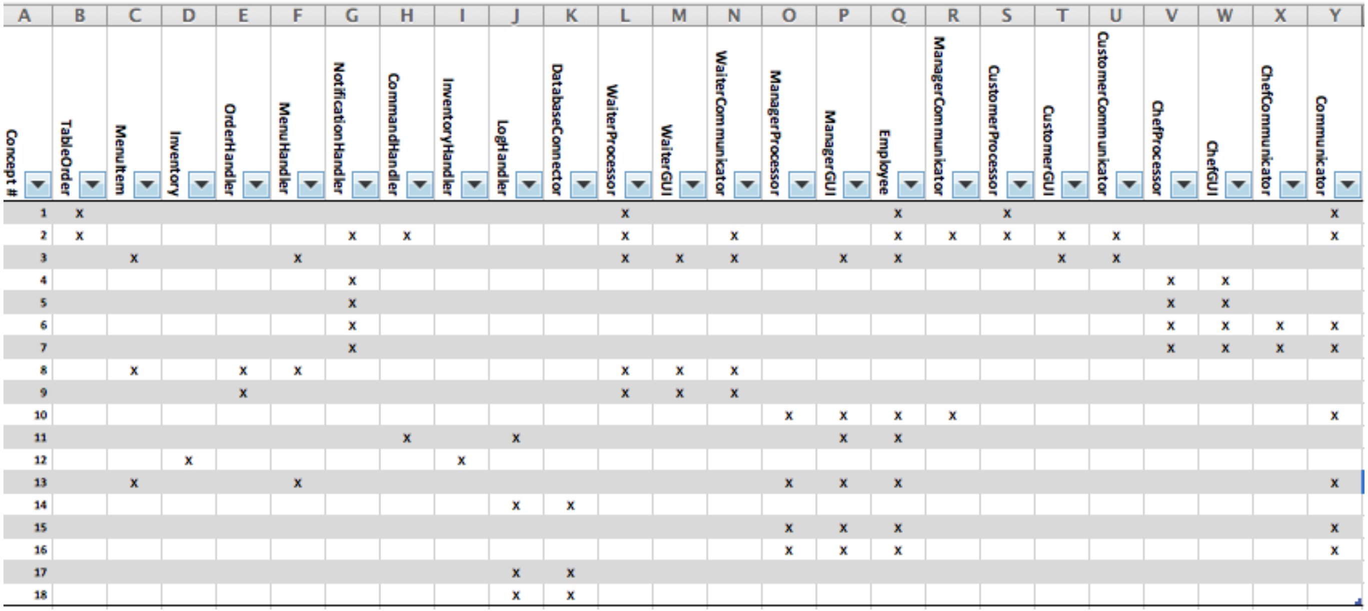
+main(args:String[0….*]):void

This method is used to initialize the GUI.

-intialize():void

This method creates the GUI.

Traceability Matrix



A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y
Concept #	TableOrder	MenuItem	Inventory	OrderHandler	MenuHandler	NotificationHandler	CommandHandler	InventoryHandler	LogHandler	DatabaseConnector	WaiterProcessor	WaiterGUI	WaiterCommunicator	ManagerProcessor	ManagerGUI	Employee	ManagerCommunicator	CustomerProcessor	CustomerGUI	CustomerCommunicator	ChefProcessor	ChefGUI	ChefCommunicator	Communicator
1	x										x					x		x						x
2	x					x	x				x		x			x	x	x	x	x				x
3		x			x						x	x	x		x	x			x	x				
4						x															x	x		
5						x															x	x		
6						x															x	x	x	x
7						x															x	x	x	x
8		x		x	x						x	x	x											
9				x							x	x	x											
10														x	x	x	x							x
11							x		x						x	x								
12			x					x																
13		x			x									x	x	x								x
14									x	x														
15														x	x	x								x
16														x	x	x								x
17									x	x														
18									x	x														

3. Architectural Styles

a. An architectural style is a set of principles that provides an abstract framework for a set of systems. The main purpose is to improve partitioning and promote design to our problems by providing detailed solutions. We will be focusing on multiple architectural styles that correspond to our project. These topics will include communication, deployment, domain and structure.

Communication: Service-Oriented Architectural Style
Service-oriented architecture enables application functionality to be provided as a set of services and creation of applications that make use of software services. they basically focus on providing a message based interaction with an application through interfaces that are applicable. Our project will have a main home system that will communicate and interact with all the PCs and tablets. However, these tablets will be autonomous since they all will have a different task and job that will notify the system to update the overall components. Services are also distributable since these portable PCs can be carried around throughout the restaurant and used whenever. Services will also share contracts when communicating and not internal classes.

Deployment: Client/Server
The client/server architecture will distribute the system that involves a separate client and server system with the overall connecting network. It describes the relationship between them whereas the client will initiate requests and wait for the reply from the system. We will use our system as the client and the tables as the server. The servers will be able to request information from the system, and the

system will distribute accordingly. Our communication protocols will also have a common language when deploying and we are looking at either C or C++.

Domain: Event Driven Design

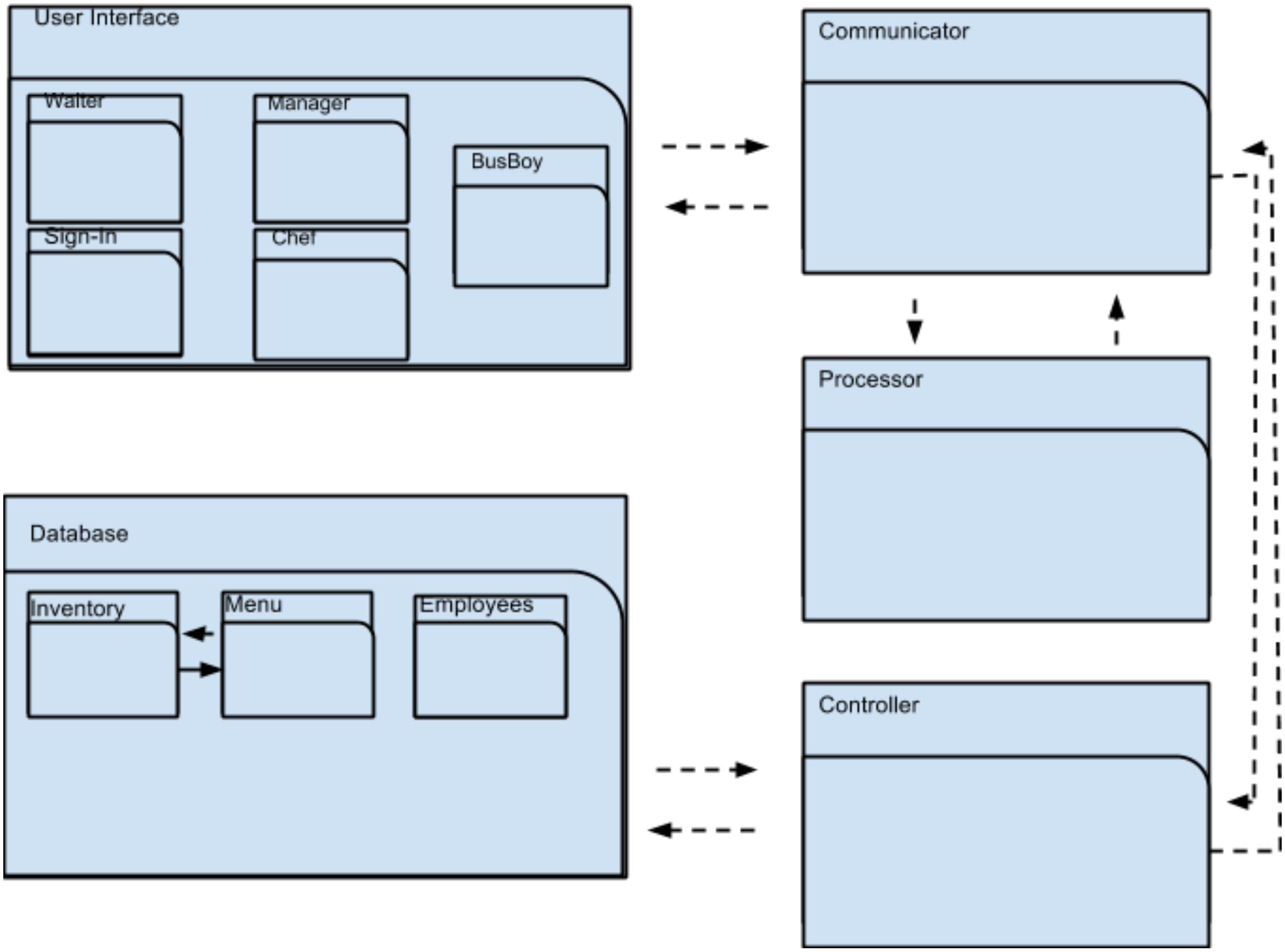
Event Driven Design is a software pattern that helps promote the production, detection, consumption, and reaction to all events. An event can be considered a significant change of an object or state of it embedded in the system. Our system will have notifications that will move the events along. For example, our chef will have notifications that will send out to the waiters when the food is ready. The waiters will be notified once the customers are seated at a certain table. The manager will be notified if an item in the inventory is about to run out or expire. All these main events and more will be crucial for the restaurant business and the event design will be sent out by the system by sending notifications.

Structure: Object-Oriented Architectural Style

Object-oriented architecture is a design pattern that divides all the responsibilities of an application or system into individual reusable objects that still maintains the data. These independent objects will communicate through our interfaces by calling methods or accessing properties of other objects by sending and receiving messages. Our structure is composed of many different tablets and PCS. Referring to abstraction, the manager tablet will have functions like Get() and Update() with the inventory and items. Encapsulation makes it easy for the tablets to delete or update items to have the newest possible update. Inheritance allows objects to be functional throughout since a single update on any machine will influence other machines by updating the system instead of one individual tablet.

b. Identifying Subsystems

Package Diagram



Based on the packaging diagram above, the packages are divided into 5 main parts. The user interface contains the 5 different interface: Chef, Waiter, Manager, Busboy and Customer Sign- In. These are the 5 different interfaces that an employee or customer may access. There is a separate package for database. This contains all the information on the menu and inventory as well as information on the employees. The database may also contain other folders. The communicator package is the accessor of information. It sends interactions from the interface to the controller. The controller delegates what the communicator can do. It accesses the database and uses the information to interact with the communicator and the processor.

c. Mapping Subsystems to Hardware

From our subsystem diagram above, our subsystem will be mapped to our hardware in a very simple way. The database will be a MySQL server that will run either on a portable or a centralized computer. The controllers will also be on this computer as well in order to bridge the gap between the user interfaces and the database. Both the database and controller will serve the back end of the process. But each user interface which will serve as the front end, such as the chef, the waiter,or the manager, will be set on individual computer or separate tablets for each user. Each of these interfaces will incorporate the necessary corresponding processor and communicator.

d. Persistent Data Storage

For our database, all the data is permanently saved and updated frequently. So, persistent data storage is essential in the transactions done in our system on a daily basis. Some transactions are customer’s orders, menu changes, and inventory changes. For our system, all the transactions depend on the table selected. Each table will hold different transactions, which go into our database. This is all done using SQL, which will maintain our database and everything does coincide with each other in some way. Our database works in a way that the applications in our system acts as the customer to the sever. This does not interact with the database directly, but simply just asks the server to perform an operation. With this built in server, our data is maintained and allows alteration of features.

e. Network Protocol

For our system, we are going to have an application which can run on a computer(for prototyping) that will be communicating back and forth with a server. We decided to go with a Microsoft SQL server since the backend will be on C++ and the front end of the computer application will also be in C++. This way we can maximize performance and compatibility by communicating locally through a network. Since we will not need to go through the internet, we can avoid TCP/IP and go straight for a Named Pipe. Named Pipes have an advantage over TCP/IP because they are usually faster for sending information, and have more free network stack resources. Named pipes are easily configured through the SQL server and the client through options and code.

Sample code to set up server and database:

```
QCoreApplication a(argc, argv);

QString servername = “LOCALHOST\\SQLEXPRESS”;    //server name
QSTRING dbname = “test”;              //database name
```

```
QSqlDatabase db = QSqlDatabase::addDatabase(“QODBC”);   //database driver
db.setConnectionOptions();              //set connections

QString dsn = QString(“DRIVER={SQL Native Client};SERVER
=%1;DATABASE=%2;Trusted_Connection=Yes;”).arg(servername).arg(dbname);
 //connection string

db.setDatabaseName(dsn);  //database name to connection string
```

f. Global Control Flow

Execution Orderness

“Why Wait” is procedure-driven. Everything executes in the following linear fashion:
When the customer first comes in, they will choose a table which is empty using the Customer Sign in PC. Once they are seated the waiter will come to get the customer’s order. The waiter will then place the order. The Chef will receive the order on the Chef’s PC and begin preparing the food. The inventory will get adjusted as needed. Once the food is ready the Chef will use the Chef’s PC to notify the Waiter PC. The Waiter will come get the food and deliver it to the customer’s table. Once the customer is finished eating the waiter will get the payment from the customer on the Waiter’s PC. Once the payment goes through, tables will be marked as dirty and the Busboy PC will call the busboy to come clean.

Time Dependency

“Why Wait” has an event-response time for the inventory alerts, but the rest of the system is a real-time system that is periodic. The procedure shown in the Execution Orderness section is what is periodic. The customers choosing a table and ordering their food, the chef preparing their food, the waiter delivering the customer’s food and taking the payment from the customer, and the BusBoy cleaning the table is all periodic. All of the previously mentioned processes are time dependent as the time that each actor takes to do their function will be taken into consideration for our system processes. Another factor where real-time plays a role is when the system estimates the amount of time the customer will have to wait for their order to be ready.

The inventory functions are also time dependent because when an ingredient comes close to expiration date the system will send out a notification to the manager. Also, if the inventory is low, a notification gets sent to the manager.

Concurrency

“Why Wait” will use multi-threading because there are multiple subsystems running independently of each other. All interactions between the subsystems are controlled through a central server. Multiple customers will be taken care of at once so multiple orders will be placed at the same time. The event where multiple customers are served will be taken care of by running the different threads through the order queue. Another situation where multi-threading would be used is when the manager is checking the inventory and updating it. The manager would have to spawn one thread for checking the inventory and another thread to handle the update request in the database. The synchronization of these threads are not necessary because they would not be working together.

Hardware Requirements

The system will need a server that would store the databases and allow communication between different subsystems. The system will also need a tablet for each of the waiter, host, manager, and chef.

Server:

Hardware	**Minimum Requirements**
Processor	Intel Xeon E5 2.3 GHz
Hard Drive	500 GB
RAM	8 GB
Network Card	10/100/1000Mbps

Tablets:

Hardware	**Minimum Requirements**
Processor	Intel i5 E5 2.4 GHz
Hard Drive	32GB
RAM	2 GB
Display	Multi-touch screen - 1024 x 760 Resolution
Network Card	802.11b/g/n Wireless LAN

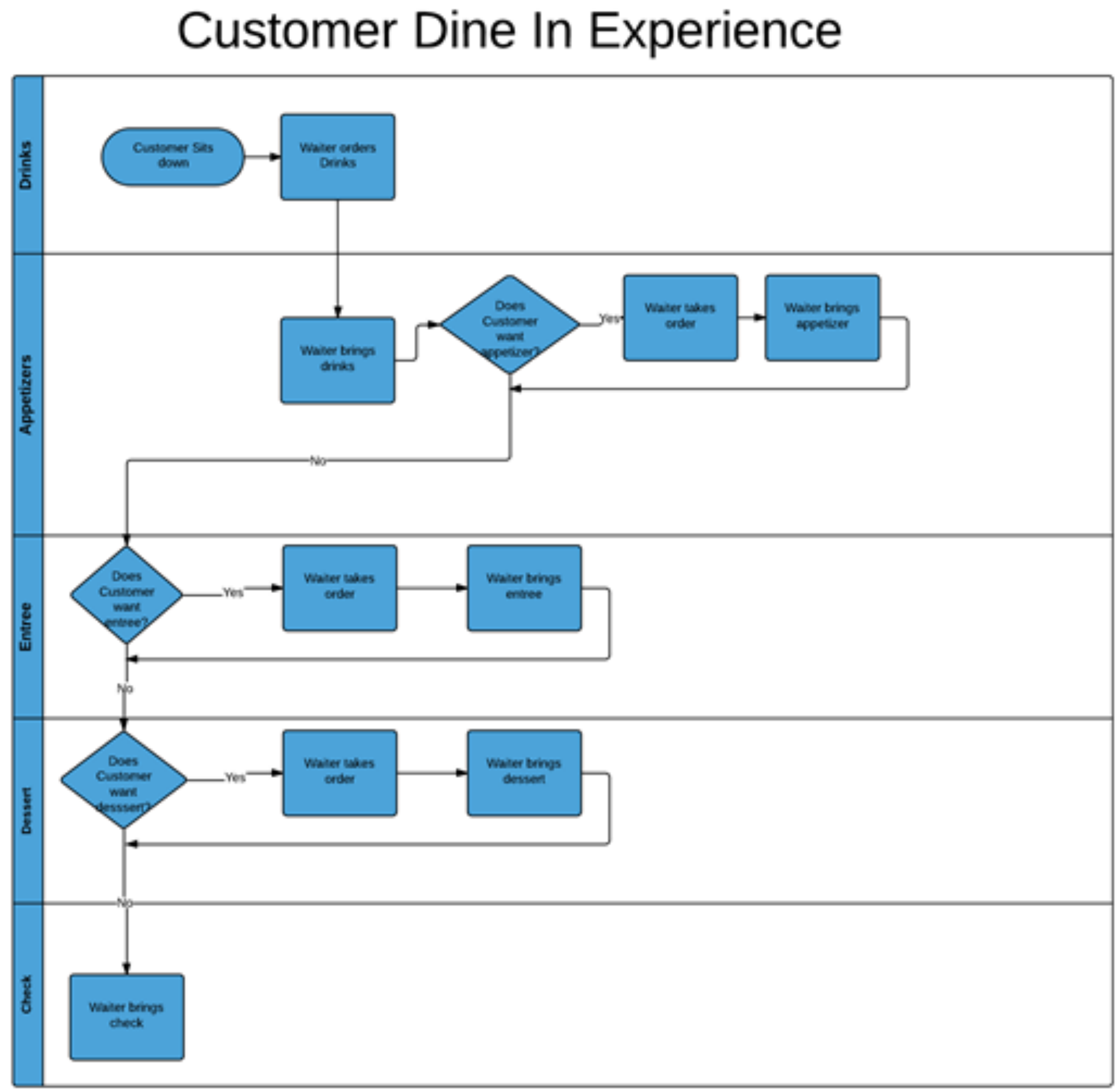
4. Algorithms and Data Structures

a. Algorithms

Waiter:

```
If (timeSinceLastVisit==8)
return notification;
```

The way this algorithm would work is if the waiter has not checked on the table in under 8min, the WaiterPC tablet will notify the waiter to check on the table. When the customer is seated, the waiter will greet the customers after the 1st minute; bring drinks after the 3rd minute; appetizers by the 8th minute if ordered; entree by the 20th minute but if no appetizers are ordered, then by 15th minute; and desserts 15 minutes after the entree; If no dessert then check within 15 after entree, otherwise 5 minutes after dessert is served



Manager:

In the manager system there are some simple algorithms being used, one of which updates the inventory as items are used or restocked. Therefore, the inventory system will increment the quantity by one or decrement it by one automatically. Another algorithm is the not that queries the database, this is used the the database management system and this communicates with the database tables themselves. Some of these algorithms include, adding an new item to the database, removing an item from the database, and updating an item's status. These database tables include, user account information, food menu information, table size and occupancy, and inventory items.

Chef:

The chef uses an algorithm for cooking orders in a queue. There is a queue within a queue as described in the mathematical section. The chef also uses an algorithm when an order is cooked to decrement the supplies as they are used. When an order is cooked, the ingredients are automatically subtracted and the database is updated.

Customer:

The customer will use a given algorithm for the seating process. The tables will have a toggle option that will show whether it is available or unavailable. The customer will pick the a table that they will like but it has to be unavailable. There will also a timer function that goes along with it that keeps going up and counting when a table is selected. It will then reset once the customer leaves.

b. Data Structures

DIfferent data structures are needed for each of the user. Array lists are used for storing data such as the items in stock or the employee information. Array lists make it easy to store data. Items can be removed and added to the list. Queues are used for prioritizing primarily by the chef and waiter. Queues are essential to make sure that orders that come first are served so that customers have shorter and more equal waiting times.

Manager

The manager would use array lists. This is because of flexibility. In an array list. Items can be deleted and added to any part of the queue therefore it would be easier to implement and easier to manipulate in the future. It is also essential in keeping track of employees who also need to be added and deleted.

Chef

The chef uses queues for making sure that orders that come first are made to order before orders that come after. This is because of performance. If an order comes earlier and is cooked later, then the customer will have a long waiting time and the waiting times of all customers will be uneven and disorganized. The strict format of the queue allows these processes to be easier.

Waiter

The main data structure for the WaiterPC is for placing the order. Each table's order will be a multi-dimensional array. I.e.

Table 1	Table 1	Table 1	Table 1
Customer 1	Customer 2	Customer 3	Customer 4
Drink	Drink	Drink	Drink
Appetizer	Appetizer	Appetizer	Appetizer
Entree	Entree	Entree	Entree
Dessert	Dessert	Dessert	Dessert

This design was used based on the flowchart shown in the algorithm section for the WaiterPC.

Customer:

The tables given for the customers will be an array list. This will make it easier for employees to add or delete tables at a given time. Selecting the tables will be a toggle or a boolean. It simply will allow a customer to select the table or not, depending on the 2 statuses of either available or unavailable.

5. User Interface Design and Implementation

During the initial interface design of our system we took careful consideration on minimizing user effort while keeping Ease-of-use in mind. Our main objective was have our system do exactly what the customer requirements entailed for each specified goals. Therefor, we have not made any significant chances to the chef, bus boy, or waiter GUI implementation. However, we feel that the manager side of the system needs some more work. The manager GUI implementation has a difficult Ease-of-use because as admin you need to make changes to the system on the fly. As far as the user effort for the manager, we feel that the system is optimized and straight to the point.

6. Design of Tests

Note For the following Test case diagrams we decided to use Spring 2013 template since we felt it was clean, easy to understand, and straight to the point. We are also going to implement some of their test cases.

Manager

For the manager system we plan on testing every function in the system besides the few we highlighted below. Since the manager has admin rights to the system, all the changes he or she makes will be reflected across all the devices. In the testing phase will be consistently adding and removing items such as employees and food items in order to make sure everything is working as intended. For algorithm testing in the case of the manager will be Inventory prediction. we plan manualing inputting dummy variables and changing dates over a long period of time in order to see how well it can predict food outcomes.

Test-case Identifier: TC - 01 **Function Tested**: addInventoryItem(inventoryItem:inventoryItem) :Boolean **Pass/Fail Critera**: the test will only pass if new item appears on list

Test Procedure	**Expected Results**
-Call Function(Pass) -Call Function(Fail)	-New item shows up on an updated inventory list -Function will return NULL value if no new item is added to the list

Test-case Identifier: TC - 02 **Function Tested**: removeInventoryItem(name:String) :Boolean **Pass/Fail Critera**: This test will only pass if item is successfully removed

Test Procedure	**Expected Results**
-Call Function(Pass) -Call Function(Fail)	-Old item does not show up on the updated inventory list -Function will return NULL value if item fails to be removed

Test-case Identifier: TC - 03 **Function Tested**: viewInventory():ArrayList **Pass/Fail Critera**: the test will only pass if a list with all inventory items appear

Test Procedure	**Expected Results**
-Call Function(Pass)	-All inventory items are return to screen

-Call Function(Fail)	-Function will return NULL value if nothing is returned

Test-case Identifier: TC - 04 **Function Tested**: viewPopularity(name:String):int **Pass/Fail Critera**: the test will only pass if the popularity stat of an item is returned

Test Procedure	**Expected Results**
-Call Function(Pass) -Call Function(Fail)	-The corresponding item will be returned to screen with its popularity -Function will return -1 value if the stat does not successfully return

Customer

The customer will have many options and functions that will implement how the restaurant system will go. They will be able to select the tables they want which will set the flow for the restaurant. There will also be a function to add or remove the tables. There will also be the status for the tables, showing if they are either available or unavailable. In addition, they can also view the wait time for each table and the time the customers are there for satisfaction.

Test-case Identifier: TC - 05 **Function Tested:** Customer::SelectTable(Table): **Pass/Fail Criteria:** The system will pass if customer selected available table, it will fail if customer selects a table that can't be selected.

Test Procedure:	**Expected Results:**
-Call Function(Pass) -Call Function(Fail)	The system will pass if customer selects correct table, it will be given to customer and the table will become not available. The system will fail if customer selects a table that is already taken, the system will return an error message requesting customer to select a different table.

Test-case Identifier: TC - 06 **Function Tested:** Customer::viewWaitTime (Menuitem m) : int throws exception **Pass/Fail Criteria:** The test passes if the correct wait time in seconds is returned from the controller for the menu item passed as a parameter.

Test Procedure:	**Expected Results:**
-Call Function(Pass) -Call Function(Fail)	Customer will be able to see correct wait time and time of customer. An error will occur and the time will be failed to sent, function will throw exception.

Test-case Identifier: TC - 07 **Function Tested:** Customer::viewStatus() **Pass/Fail Criteria:** The system will pass if it shows the correct status, it will fail otherwise.

Test Procedure:	**Expected Results:**
-Call Function(Pass) -Call Function(Fail)	Customer will be able to see correct status of the table. An error will occur and the status will be failed to sent, function will throw exception.

Test-case Identifier: TC - 08 **Function Tested:** addTable(Table) **Pass/Fail Criteria:** The test will pass if a table is added correctly, otherwise an error will occur it if it doesn’t.

Test Procedure:	**Expected Results:**
-Call Function(Pass) -Call Function(Fail)	The new table will be added correctly and it will show up on the system. An error will occur and the table won’t be added

	and the system will ask for you to try again or quit.

Test-case Identifier: TC - 09 **Function Tested:** deleteTable(Table) **Pass/Fail Criteria:** The test will pass if a table is deleted correctly, otherwise an error will occur it if it doesn't.

Test Procedure:	**Expected Results:**
-Call Function(Pass) -Call Function(Fail)	The current table will be deleted correctly and it will not show up on the system. An error will occur and the table won't be deleted and the system will ask for you to try again or quit.

Test-case Identifier: TC-10 **Use Case Tested:** Waiter::notification (Table) **Pass/Fail Criteria:** The test passes if the system is able to notify the waiter according to our algorithm

Test Procedure:	**Expected Results:**
-Call Function(Pass) -Call Function(Fail)	WaiterPC tablet gets a notification if customer is at table and waiter has not checked in 8 minutes Waiter tablet doesn't get a notification because no customers at table or an error occurred in the system

Test-case Identifier: TC-11 **Use Case Tested:** Waiter::AddItem (Menuitem m) **Pass/Fail Criteria:** The test passes if the item is added to the queue

Test Procedure:	**Expected Results:**
-Call Function(Pass)	The correct data is sent through and the item that

-Call Function(Fail)	needs to be added goes on the queue of the ChefPC and as well as an added item on the specific table for the waiter to view Item is not available or an error occurred in the system

Test-case Identifier: TC-12
Use Case Tested: Waiter::DeleteItem (Menuitem m)
Pass/Fail Criteria: The test passes if the item is deleted from the queue

Test Procedure:	**Expected Results:**
-Call Function(Pass) -Call Function(Fail)	The correct data is sent through and the item that needs to be removed comes off on the queue of the ChefPC and as well as deleted on the specific table for the waiter to view Item is not available or an error occurred in the system

Test-case Identifier: TC-13
Use Case Tested: Waiter::ViewQueue () : ArrayList<MenuItem>
Pass/Fail Criteria: The test passes if the menu items are viewed from each table on the queue for the Waiter

Test Procedure:	**Expected Results:**
-Call Function(Pass) -Call Function(Fail)	The correct data is sent through and the items ordered by each customer on each table can be viewed on the queue for the Waiter Table is empty or an error occurred in the system

Chef

The chef will have basic actions and functions to utilize in the testing phase. The chef will only need to communicate with the waiter in order to receive the order. Within the testing phase, the chef will be able to also view the order queue of the current orders. The chef will also be able to add or delete items within an order in accordance to the customers preferences. However, the chef can ultimately determine when the order has been completely prepared and be ready to be taken to the customer using the function of itemfinished as call to the waiter that the order is done.

Test-case Identifier: TC-14
Use Case Tested: Chef::ViewQueue () : ArrayList<MenuItem>
Pass/Fail Criteria: The test passes if the menu items are viewed from each table on the queue for the Waiter

Test Procedure:	Expected Results:
-Call Function(Pass)	The correct data is sent through and the items ordered by each customer on each table can be viewed on the queue for the Waiter
-Call Function(Fail)	Table is empty or an error occurred in the system

Test-case Identifier: TC-15
Use Case Tested: Chef::ItemFinished () : ArrayList<MenuItem> : Boolean throws exception
Pass/Fail Criteria: The test passes if the item is removed from temporary storage and passed to the waiter via the controller

Test Procedure:	Expected Results:
-Call Function(Pass)	Correct data to be sent is passed, function returns true if the controller successfully passes the MenuItem to the waiter to be delivered.
-Call Function(Fail)	MenuItem incorrect or controller error, function returns false. Message fails to be sent, function throws exception

Test-case Identifier: TC - 16 **Function Tested:** Chef::AddOrder (TableOrder o) : Boolean **Pass/Fail Criteria:** The test passes if the menu items in the order pass in as an argument are successfully scheduled into the chef queue.

Test Procedure:	**Expected Results:**
-Call Function(Pass)	Correct data to be sent is passed, function returns true if all the menu items part of the table order passed are scheduled successfully.
-Call Function(Fail)	If functions fails to schedule all the orders, returns false

Test-case Identifier: TC - 17 **Function Tested**: Chef::RemoveOrder (TableOrder o) : Boolean **Pass/Fail Criteria**: The test passes if the menu items in the order pass in as an argument are successfully removed from the chef queue.

Test Procedure:	**Expected Results:**
-Call Function(Pass)	Correct data to be sent is passed, function returns true if menu item is removed.
-Call Function(Fail)	Function returns false, if menu item is incorrect or cannot be removed.

7. Project Management and Plan of Work

a. Merging Contributions From Individual Team Members

In order to merge our work together and be organized, we made a google document. This was the best way since we are able to work together in different locations and have our work saved at the same time on one final document we are able to edit together. Some issues that we encountered were that our format would be a little different and we would be unorganized since our parts would be at different locations. This problem was not that hard to solve. We fixed the format issue by agreeing on a specific font and spacing. We fixed the unorganization part by labeling everything beforehand when making the template so we are able to put the correct parts in the same place.

b. Project Coordination and Progress Report

History of Work

January 21st- January 29th

Our team decided to take on the restaurant automation project and created the proposal of how we would go about tackling this project and how we would decide to implement this. We began by reading through past projects and understanding where they did well, and where they also had weaknesses, and we decide to create a system that would build upon those weaknesses

February 1st- February 23rd

We received feedback on how to improve our proposal and how we could plan better for our project as well. Our aim was to build upon the new proposal and incorporate those new ideas into our first report. A theoretical model of our system was built using the report guidelines given, incorporating Customer Statement of Requirements, Glossary of Terms, Functional Requirements, Effort Estimation, and Domain Analysis. We divided the work evenly and pieced together the report at the end, submitting our report on February 23rd

February 24th- March 19th

The team then began planning for the second report, building on what we learned from drafting our first report on the specification of our system. This time we would focus on the design aspect of the system. Through the drafting of the second report, we began to develop part of the system that we would be using during the first demo. Through the Interaction Diagrams, the Class Diagrams and Interface Specification, System Architecture and System Design, Algorithm and Data Structures, User Interface Design and Implementation, and Design of Tests, we drafted our second report and began to build our system. We focused on building the server and the GUI and divided the work evenly between all team members.

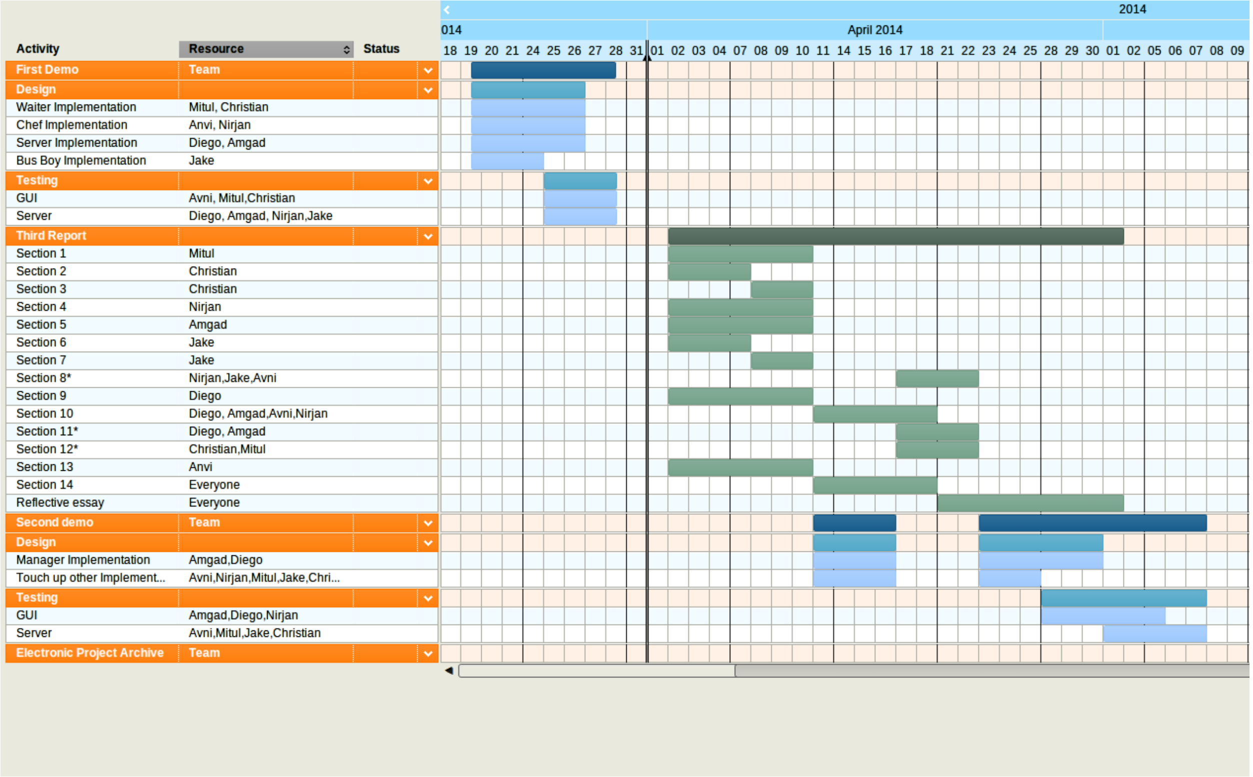
Current Status

Our system is currently being developed, the building being separated into two components, the server and the GUI. We have divided the overall team into two teams to work separately on either component. As of right now, we do not have any functional components, but we are focusing our maximum effort into completing parts of the server and the GUI in time for our first demo presentation

Future Work

Currently, we are in the middle of developing a working version of our server and GUI for the first demo presentation. For the demo, we hope to have a server and GUI that interact with each other, and communicate with other modules as well, such as the chef and waiter modules. Achieving this communication is a top level priority because that communication is what will allow the entire system to work as a whole. But it also poses a problem as the modules will be worked on with separate groups, so the modules won't work together at first due to maybe different programming languages or even errors in the coding of the modules. But ultimately, we want to have our server and GUI combine with all of the modules and interact as a single system.

c. Plan of work



Activity	Resource	Status
First Demo	**Team**	
Design		
Waiter Implementation	Mitul, Christian	
Chef Implementation	Anvi, Nirjan	
Server Implementation	Diego, Amgad	
Bus Boy Implementation	Jake	
Testing		
GUI	Avni, Mitul,Christian	
Server	Diego, Amgad, Nirjan,Jake	
Third Report		
Section 1	Mitul	
Section 2	Christian	
Section 3	Christian	
Section 4	Nirjan	
Section 5	Amgad	
Section 6	Jake	
Section 7	Jake	
Section 8*	Nirjan,Jake,Avni	
Section 9	Diego	
Section 10	Diego, Amgad,Avni,Nirjan	
Section 11*	Diego, Amgad	
Section 12*	Christian,Mitul	
Section 13	Anvi	
Section 14	Everyone	
Reflective essay	Everyone	
Second demo	**Team**	
Design		
Manager Implementation	Amgad,Diego	
Touch up other Implement...	Avni,Nirjan,Mitul,Jake,Chri...	
Testing		
GUI	Amgad,Diego,Nirjan	
Server	Avni,Mitul,Jake,Christian	
Electronic Project Archive	**Team**	

d. Breakdown of Responsibilities

WaiterPC Responsibilities:	Mitul	Christian	Amgad	Diego	Jake	Nirjan	Avni
Customer Statement of Requirements (CSR)	X	X	X	X	X	X	X
System Requirements	X	X	X	X	X	X	X
Functional Requirements	X	X	X	X	X	X	X

Specification							
User Interface Specification	X	X	X	X	X	X	X
Domain Analysis	X	X	X	X	X	X	X
Interaction Diagrams	X	X	X	X	X	X	X
Class Diagram and Interface Specification	X	X	X	X	X	X	X
System Architecture and System Design	X	X	X	X	X	X	X
Algorithms and Data Structure	X	X	X	X	X	X	X
User Interface Design and Implementation	X	X	X	X	X	X	X
Design of Tests	X	X	X	X	X	X	X
Project Management	X	X	X	X	X	X	X
Plan of Work	X	X	X	X	X	X	X
References	X	X	X	X	X	X	X

ManagerPC Responsibilities:	Mitul	Christian	Amgad	Diego	Jake	Nirjan	Avni
Customer Statement of Requirements (CSR)	X	X	X	X	X	X	X
System Requirements	X	X	X	X	X	X	X
Functional Requirements Specification	X	X	X	X	X	X	X
User Interface Specification	X	X	X	X	X	X	X

Domain Analysis	X	X	X	X	X	X	X
Interaction Diagrams	X	X	X	X	X	X	X
Class Diagram and Interface Specification	X	X	X	X	X	X	X
System Architecture and System Design	X	X	X	X	X	X	X
Algorithms and Data Structure	X	X	X	X	X	X	X
User Interface Design and Implementation	X	X	X	X	X	X	X
Design of Tests	X	X	X	X	X	X	X
Project Management	X	X	X	X	X	X	X
Plan of Work	X	X	X	X	X	X	X
References	X	X	X	X	X	X	X

ChefPC Responsibilities:	Mitul	Christian	Amgad	Diego	Jake	Nirjan	Avni
Customer Statement of Requirements (CSR)	X	X	X	X	X	X	X
System Requirements	X	X	X	X	X	X	X
Functional Requirements Specification	X	X	X	X	X	X	X
User Interface Specification	X	X	X	X	X	X	X
Domain Analysis	X	X	X	X	X	X	X
Interaction Diagrams	X	X	X	X	X	X	X
Class Diagram and Interface Specification	X	X	X	X	X	X	X
System Architecture and	X	X	X	X	X	X	X

System Design							
Algorithms and Data Structure	X	X	X	X	X	X	X
User Interface Design and Implementation	X	X	X	X	X	X	X
Design of Tests	X	X	X	X	X	X	X
Project Management	X	X	X	X	X	X	X
Plan of Work	X	X	X	X	X	X	X
References	X	X	X	X	X	X	X

BusBoyPC Responsibilities:	Mitul	Christian	Amgad	Diego	Jake	Nirjan	Avni
Customer Statement of Requirements (CSR)	X	X	X	X	X	X	X
System Requirements	X	X	X	X	X	X	X
Functional Requirements Specification	X	X	X	X	X	X	X
User Interface Specification	X	X	X	X	X	X	X
Domain Analysis	X	X	X	X	X	X	X
Interaction Diagrams	X	X	X	X	X	X	X
Class Diagram and Interface Specification	X	X	X	X	X	X	X
System Architecture and System Design	X	X	X	X	X	X	X
Algorithms and Data	X	X	X	X	X	X	X

Structure							
User Interface Design and Implementation	X	X	X	X	X	X	X
Design of Tests	X	X	X	X	X	X	X
Project Management	X	X	X	X	X	X	X
Plan of Work	X	X	X	X	X	X	X
References	X	X	X	X	X	X	X

8. References

"Concepts: Requirements." Concepts: Requirements. Polytechnique Montreal, 2012. Web. 5 Feb. 2014. -(Used for Non-Functional Requirements)

Nick Leshi. (2010). Good Restaurants Come and Go. Available: http://open.salon.com/blog/kikstad/2010/06/25/good_restaurants_come_and_go. Last accessed 8th Feb 2014. -(Used for Cover picture).

Group#1 Spring 2013. (2013). Inventory usage rate estimation and runout date estimation. Auto-Serve. 1 (2), 75-116.

Group#1 Spring 2013. (2013). Inventory usage rate estimation and runout date estimation. Auto-Serve. 1 (2), 129-151.