# Auto-Serve

Restaurant Automation

Getting Served! Now Automatic.

*Group 1*

*All members contributed equally.*

*Prem Patel*
*Sai Kotikalapudi*
*John Bartos*
*Scott Xu*
*David Shen*
*Joshua Devasagayaraj*

## Table of Contents

# 1. Customer Statement of Requirements

Many restaurants today still use the same basic methods from years ago to handle orders from customers. Often times this leads to complicated coordination of activities between chefs and waiters. This also means that other tedious tasks, such as inventory checks, fall to the managers to perform. This project seeks to introduce automation in privately-owned restaurants to alleviate some of the problems associated with current restaurant management practices.

## 1.1 Problem

After researching current restaurant practices and work from previous groups, we narrowed our focus to these problems:

### 1.1.1 Keeping Inventory and Determining When to Restock

Managers have the incredibly tedious role of keeping inventory and deciding whether or not to order more supplies for the restaurant. A survey taken by a previous group who interviewed the manager at a Buffalo Wild Wings says,

“[The manager] hates the fact that he has to go in and manually check each ingredient and see what you need more of for the next shipment by paper and hand.” - Group 2, 2011

It was evident that the burden of the accounting work fell on the manager. We know that this burden could be partially reduced, if not completely eliminated, through automation.

Furthermore, after accounting for the current inventory, the manager must decide whether to restock or not. This decision requires the manager to estimate how long the restaurant’s remaining supplies will last. This estimation requires knowledge of past usage rates and other information about the restaurant’s past performance, and is not a trivial calculation; it has to take into account past usage rates and predict factors which will affect future usage, such as seasonal changes in demand and upcoming holidays.

Lastly, managers would need to know if their inventory was getting too low. If it falls below a certain threshold and they do not catch it, it would lead to costly results for the restaurants. At the very least, it would lead to a loss in profits and poorer customer ratings.

Software assistance could alleviate many of the above issues and boost restaurant efficiency by handling the inventory system autonomously, with minimal user input after the system is fully operational.

### 1.1.2 Keeping Diners Informed About Their Wait Time

One common thing that we noticed was that customers rarely know how long their food will take to finish. This was evident from our personal experience: waiters can give rough estimates based on what they know about the kitchen’s current state, but at best those are still rough guesses.

Common consequences of being left in the dark are feelings of boredom or uncertainty, leading to the customer thinking, “is there time to step outside for a cigarette, or to the bathroom? Should I order an appetizer to make the wait more bearable?” Customers become upset when their food arrives later than expected since they could have ordered appetizers to soothe their hunger.

Another potential problem is the delay between individual dishes being finished in the kitchen and then being sent to a table. What if one dish takes much longer than the rest? If a diner wants his food as soon as possible -- that is, without waiting for the table’s other orders -- he should be informed of when it’s finished as well as given the ability to request its early delivery.

### 1.1.3 How to Cook and Deliver Menu Items that Customers Order

Currently, many restaurants have chefs cook orders on a first come first serve basis. Many members of our group have seen the classic pen and paper method of organizing incoming orders at restaurants they have gone to.

Typically, the waiter takes an order and hangs the description of the order at the chef counter, where the chefs take and prepare orders one by one. After cooking, the food is placed in a “ready area” with the order description.

Waiters additionally have the job of searching for their respective orders and checking that all the food is ready; with no system to directly notify waiters, this forces them to periodically return to the kitchen for essentially no reason.

After looking at this process, we knew there was a lot of room for improvement. The overall efficiency of the restaurant and consequently customer satisfaction can be improved.

### 1.1.4 Tracking Menu Item Popularity and The Menu Item Rating System

Restaurant managers can sometimes have a difficult time determining what menu items are popular and what items are infrequently ordered. Customers too want to know what dishes are the best and what dishes to avoid. For most small restaurants, advertising is achieved through word of mouth. New customers of the restaurant usually hear reviews from their peers.  Their peers usually recommend the restaurant because of a certain menu item that they liked or because the service provided by the restaurant is excellent. This influences what choices the customers have.

When designing or updating the menu, restaurant managers have the difficult dilemma of adding great new dishes and removing stale, unpopular old dishes. By making customer satisfaction information available in the form of popularity and ratings numbers, the manager has a source of hard data and a unified view of the customer preferences to base his decisions off of.

A list of menu items sorted by popularity and/or rating would serve everyone well: instead of asking waiters (who have a smaller role in our automated restaurant) for recommendations, customers can view the popular dish list directly. Instead of trying to guess what dishes are most and least popular, managers can view this information directly and easily and make better informed decisions.

### 1.1.5 Things to Note before Reading

When developing our solutions to these problems, we came across a few issues.

#### 1.1.5.1 Menu Items vs. Dishes

When deciding the names of menu items that customers order, we were debating between calling them “menu items” or “dishes”. In the customer’s point of view, they can be seen as menu items. However, in the chef’s point of view, they can be seen as dishes.

There are times in this document when the terms “menu item” and “dish” are both used interchangeably. Both terms refer to the same concept; in most places, “menu item” will be used, but in cases where “dish” is more appropriate it is used instead.

#### 1.1.5.2 Table Order vs. Table

When we designed the queue for waiters who had to deliver dishes back to the customers, we initially decided to represent the table as a “table” that represented the people who ordered many items from the same table. We realized that this was a poor decision since tables might order again. Therefore, for clarity, we changed the naming convention from tables to table orders (or just order) where the table order just represents the composite order for all people at the table. To represent the actual table that the customers are seated at, we just use a table ID number since the actual table need not be a concept, just an attribute.

## 1.2 Glossary of Terms

Many of the terms you see here can be understood as you read the report. They are also listed here for clarity and formality.

### 1.2.1 Technical Terms

**Order Queue** - The queue of menu items, ordered by customers that are ready to be cooked by the chefs.

**Ready Queue** - The queue of menu items that are ready to be delivered to the customers.

**Wait Queue** - The queue of menu items that are currently being prepared by the chef.

**Order** - A concept that represents the list of items that the customer ordered.

**Table** - concept that represents the physical table that customers are seated at.

**Table Order** - concept that represents the actual composite order of all the customers seated at a certain table.

**Inventory System** - an electronic book keeping of the current inventory in the restaurant. This includes its raw ingredients, current menu items, etc.

**Scheduling Policy** - a predefined set of rules to determine where in the queue or line the next item should go.

**Terminal** - a part of the restaurant where a respective user is working. E.g. The kitchen where the chef is working can be called the "Chef Terminal"

**Module** – Same thing as terminal described above but refers more to the Applications on the hardware.

### 1.2.2 Non-Technical Terms

**Chef -** Cooks all the food in the restaurant.

**Manager** - Manages the inventory, orders more supplies, and deals with overall management and finances of the restaurant.

**Waiter** - Handles delivering food and

**Customer** - the person who the service provided by the restaurant is being given too.

**Ingredient** - A food that is used to create a menu item. e.g lettuce, carrots, etc.

**Menu Item** - The food that that is listed on the menu given to customers to choose from. e.g various burgers, pasta, etc.

**Dish** - Equivalent to the menu item, but used when describing queuing related to chefs and waiters for simplicity and is more appropriate.

# 2. System Requirements

## 2.1 Proposed Solution

Our solution focuses on the problems we highlighted in the first section. We have left out trivial elements of the system such as login, ordering items, indicating dirty tables, etc. These items were already done by previous groups and their reports are applicable. We have focused on our core ideas that make our product genuine and worthwhile.

### 2.1.1 Inventory System

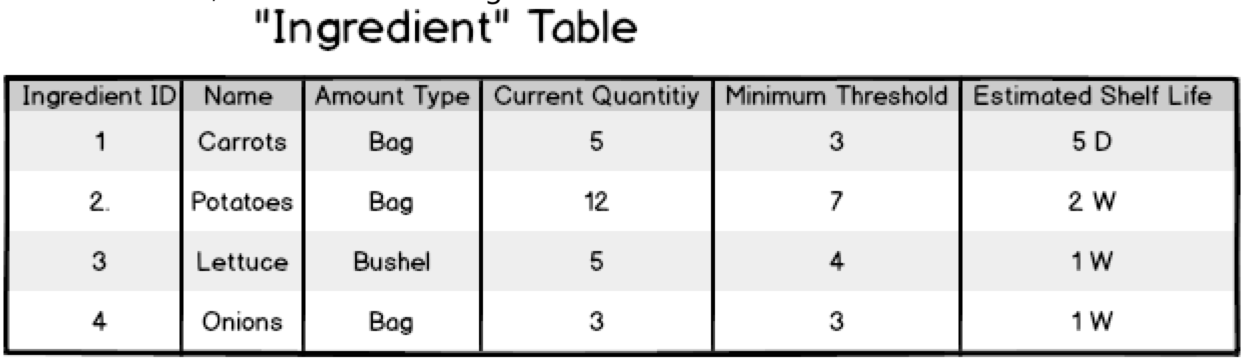
To combat the problems involved with keeping inventory and determining when to restock, we set out to design a system that takes as much of the burden as possible off of the manager.

With our smart inventory system and prediction algorithms, we can effectively reduce the amount of times the manager has to physically check to stockroom for ingredients and partially automate the restocking process.

To implement this inventory system, we chose to use a database. MYSQL seemed the most cost effective choice since it is the most popular and freely available. Since it is a medium sized restaurant, it will suit our needs.  Not only can the database hold information about individual ingredients, but we can create tables in the database to reference these ingredients so that we can store menu items to represent containers.

In database terminology, a **table** represents a matrix of r rows and columns that is serialized and stored as data depending on the type of database.

In the database, our table for raw ingredients can be visualized like this:



"Ingredient" Table

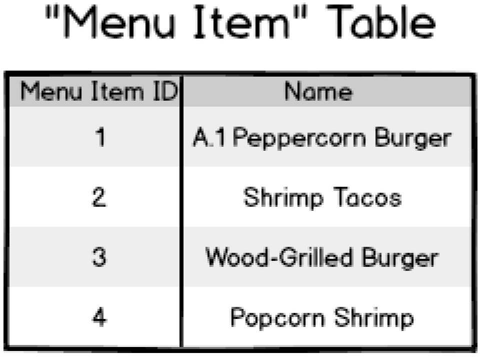
| Ingredient ID | Name | Amount Type | Current Quantitiy | Minimum Threshold | Estimated Shelf Life |
|---|---|---|---|---|---|
| 1 | Carrots | Bag | 5 | 3 | 5 D |
| 2. | Potatoes | Bag | 12 | 7 | 2 W |
| 3 | Lettuce | Bushel | 5 | 4 | 1 W |
| 4 | Onions | Bag | 3 | 3 | 1 W |

Figure 1: This is Ingredient table as represented in the database.

**Ingredient ID** represents the identification of the ingredient inside the table to relative to the other ingredients. **Name** is the name of the ingredient. **Amount Type** is the type of quantity that the ingredient is measured in. **Minimum Threshold** is the manager specified minimum amount of this items that should be in the inventory. **Estimated Shelf Life** is the estimated shelf life of the ingredient or the amount of time that the ingredient can be stored before it goes bad.

When the manager issues an order for raw ingredients to the supplier, this table can be automatically updated when the order is verified by the manager. As long as the supplier is a trustworthy source, the inventory will be correctly updated without manager intervention.

Furthermore, a table for menu items will hold all the menu items available to the customer.

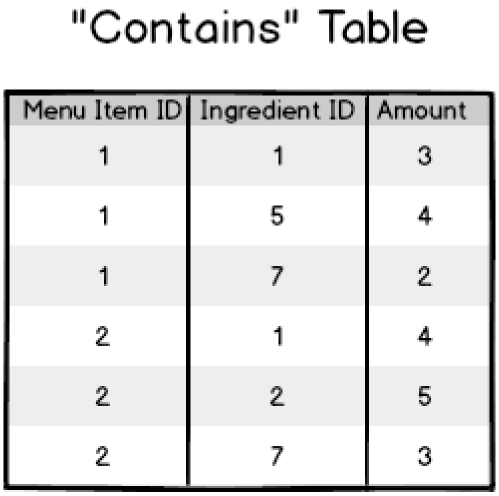


"Menu Item" Table

| Menu Item ID | Name |
|---|---|
| 1 | A.1 Peppercorn Burger |
| 2 | Shrimp Tacos |
| 3 | Wood-Grilled Burger |
| 4 | Popcorn Shrimp |

Figure 2: This is the Menu Item table as represented in the database.

To store mapping between menu items and ingredients, we create an additional table called “Contains” to store the **one-to-many relationship** that menu items have with ingredients.



"Contains" Table

| Menu Item ID | Ingredient ID | Amount |
|---|---|---|
| 1 | 1 | 3 |
| 1 | 5 | 4 |
| 1 | 7 | 2 |
| 2 | 1 | 4 |
| 2 | 2 | 5 |
| 2 | 7 | 3 |

Figure 3: This is Contains table as represented in the database.

When chefs cook menu items from the menu, the inventory system can be automatically updated to show the remaining inventory after cooking each dish. Of course, when making each dish, there is going to be some error in measuring by the chefs and the total amount of inventory usage will have a small percent error.  We discuss this issue later.

Since the system is aware of each menu item in the menu, the amount of times the menu item is ordered, and the contents of each menu item, the system can keep a real-time measure of the amount of ingredients in the inventory.

Having the system keep track of inventory leads to a plethora of features that we elaborate on later on such as future predictions of inventory requirements and knowing how much of an item we can produce.

One thing we must not forget is that the manager will always have access to the inventory system in the event that he must manually change the stock.

### 2.1.2 Inventory Usage Prediction

To streamline the management process, our system has the ability to predict ingredient usage rates. This feature solves the issue of the manager having to use historical data about the restaurant in his decision of whether to restock.

This is a data mining problem in which we take usage data over a lengthy period of time and develop predictions of how much inventory will be used and how much should be ordered based on the season, holidays, weekday, etc.

As we lack experience in this area, we decided to start out with a relatively simple approach to the problem. At the suggestion of our advisor, we also investigated more complex autoregressive, moving-average, and autoregressive-moving-average or ARMA models, but without the proper background in signal processing and statistics we found these models too difficult to implement.

A detailed description of our algorithm is included further in this document, under "Mathematical Models".

### 2.1.3 Inventory Alerts

Another feature of the inventory system that introduces novelty are alerts. This feature alleviates the issue involved with managers having to determine when inventory falls below a certain threshold.

Alerts are triggered when inventory of items fall below certain limits. These limits though obvious it may seem, are not just the bare minimum to cook a menu item but rather is threshold set by the manager.  Again when group 2 in 2011 interview Buffalo Wild Wings and asked the manager how often he restocks, he said,

"Every Wednesday and Saturday regardless of demand. Always have a surplus."

After reading this, we found it appropriate to improve this procedure and not only to automatically determine when it is best to restock, but also send the appropriate demand to the supplier himself (with manager approval, of course).

Our design of the alert system follows:

We use the predictive ingredient usage model to estimate when an ingredient falls below its critical stock threshold. The manager can set these thresholds based on demand and perishability, but in general it should be no less than 10% of the usage in the time period between restocks or the amount of the restocking order.

By recursively applying the equation we can estimate ingredient usage for an arbitrary day. First we calculate usage for day n+1 and subtract it from our current stock level. If it is below the threshold, we send the alert. If it is not, we apply the formula for day n+2 using our estimate for day n+1 as an input; if the total usage on day n+1 and n+2 cause stock levels to fall below the critical threshold, we send the alert. If not, we iterate once more; this process continues until it reaches the day when the stock falls beyond the threshold.

In the final system, we will most likely estimate usage extremely conservatively to prevent any item from going out of stock. With time, the restaurant manager can manually update/lower the thresholds and other parameters.

#### Customer Wait Time Estimation

Our system will estimate the wait time for a table order using the existing menu item queue that exists in the kitchen terminal. Each dish in the queue is associated with a table; by checking each dish for a given table, we can determine the expected finish time of the last dish and use it to determine the total wait time for the table. For tables which request dishes be delivered as they are finished, we will display the estimated wait for each dish instead of for the entire table.

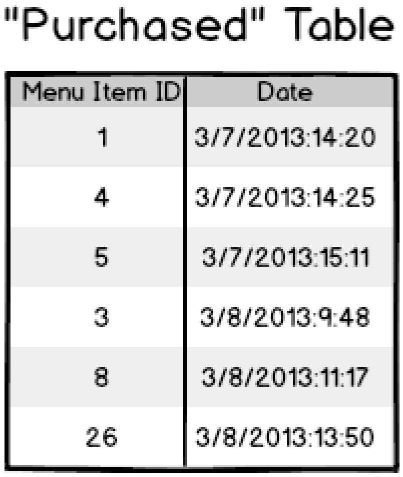
The detailed algorithm is written in the mathematical model.

### 2.1.4 Food Popularity

To solve the problem of determining what items are popular on the menu and what items are not, we have designed a new way so that both the manager and the customer will be able to know what the "hot items" on the menu are.

Since, we are using an inventory system; the task of providing popular items to the manager becomes trivial. By knowing which menu items customer's order, and the number that are ordered every day, the manager can be provided a clear cut overview of the most popular items in the current day, week, month, season, etc.

To determine food popularity we need keep track of what items are being bought every day. As we are using a database, we can add another table to keep track of this.



"Purchased" Table

| Menu Item ID | Date |
|---|---|
| 1 | 3/7/2013:14:20 |
| 4 | 3/7/2013:14:25 |
| 5 | 3/7/2013:15:11 |
| 3 | 3/8/2013:9:48 |
| 8 | 3/8/2013:11:17 |
| 26 | 3/8/2013:13:50 |

Figure 4: This is Purchased table as represented in the database.

This "Purchased" Table will be updated with the date of every order that is purchased. This way, it is simple to find the amount of times a menu item is ordered within a certain time frame.

For instance, if we want to find the most popular items last week. We just need to tabulate the number of times each item was purchased during that week.

Since this is just a simple search and count procedure, it is not needed to be described, and therefore not needed to be shown in the mathematical model.

Similarly, the system can also provide popular items to customers; however some customers may not trust statistics that the restaurant generates. The customer may think that the restaurant may be trying to sell more of one menu item over another.

Therefore, we designed a rating system so that customers can rate various menu items that they have eaten and write their own comments about items. Similar to Amazon.com's rating system, the rating system is designed so that customers can "like" other customers' review and rating so that these higher quality reviews will be higher up in the list of reviews that the customer may read for each menu item.

Since, the rating system is designed similar to Amazon.com, there is no need to go further into detail about it.

### 2.1.5 Chef's Interface and Menu Item Queuing

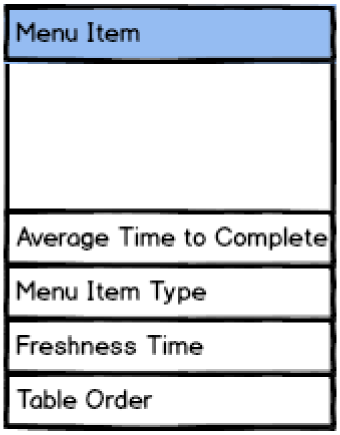
As we focused on efficiency and time saving in the automation process, one realization was that chefs can prepare foods faster if they are given similar foods that can be cooked together or in parallel. One example of this is having a chef cook two burgers and then a cheesecake rather than have a chef cook a burger, then a cheesecake, and then back to a burger. This essentially cuts the time by the length of one burger. When the chef gets the ingredients for one burger, he may also get the ingredients for the other burger and cook them in parallel.

Similar to operating system design, this problem becomes similar to that of scheduling and prioritizing processes, however in this case the processes are menu items to be cooked, and our cpu is the chef.

Keeping this in mind, we designed an optimal scheduling algorithm based on a priority queue where the new menu item will be given a priority based on what's currently cooking, and what's currently on the queue. Of course, menu items cannot be preempted because no chef would stop halfway in making a menu item as that would be wasteful and sometimes ruin the menu item in certain cases. Thus we focused on non-preemptive scheduling of menu items.

Our solution to the problem can be visualized as follows:

Each menu item on the queue for chefs can be modeled as a compound data structure. There are many parts to the menu item structure but I will only lay out the parts that pertain to the chef's queue.



Menu Item

Average Time to Complete

Menu Item Type

Freshness Time

Table Order

Figure 5: These are the attributes of the Menu Item that pertain to queuing.

Here we focus on three properties of the menu item,

**Average time to complete:** The average total time that the chef takes to cook the menu item in questions, this time must be calculated based on real data that the restaurant takes when operating under normal conditions. In our system, we assume that this data is available for us.

**Menu item Type:** The category of menu item that the menu item falls under.

**Freshness Time:** This is the time that the dish can be kept warm and still retain its freshness

**Table Order:** This is a reference to the table order that this menu item belongs to. The table order is described in detail in the section involving queuing with waiters. A table order of zero means that the item does not belong to a table order. This is used later on in queuing for waiters.

Here is the basic scenario of when a burger will move up the queue.

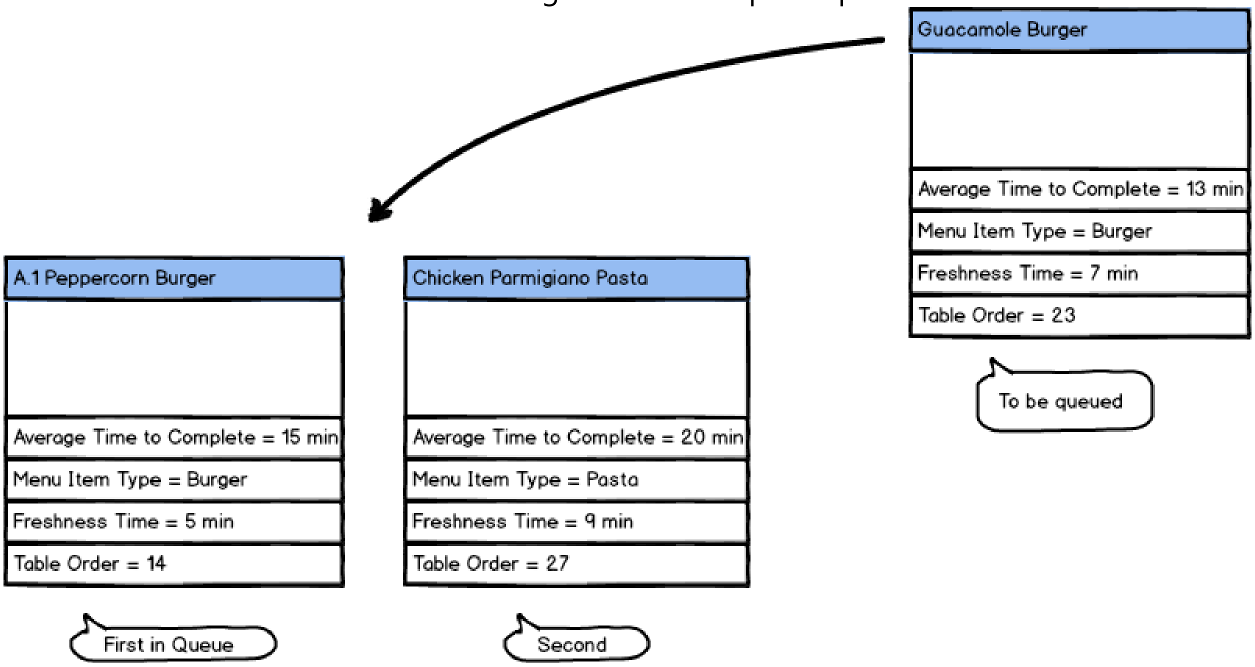


Figure 6: The guacamole burgers get queued with the A.1 peppercorn burger.

In this situation, since the guacamole burger is a burger, it can be queued together with the peppercorn burger, allowing the chef to make these burgers in parallel and essentially cutting the total turnaround time by the time to complete of the Guacamole Burger. One restriction to this is that the Guacamole Burger is only queued together because it average time to complete is less than the average time to complete of the

Peppercorn Burger. If the Guacamole Burger had an average time to complete greater than that of the Peppercorn Burger, it would not be queued since that would add time additional time to the queue that would increase the wait time of the pasta. This is not desirable since that would increase the wait time of a customer that ordered the pasta caused by someone who ordered after him. This "customer-first" approach is the key among restaurants and it's maintained in this queuing policy.

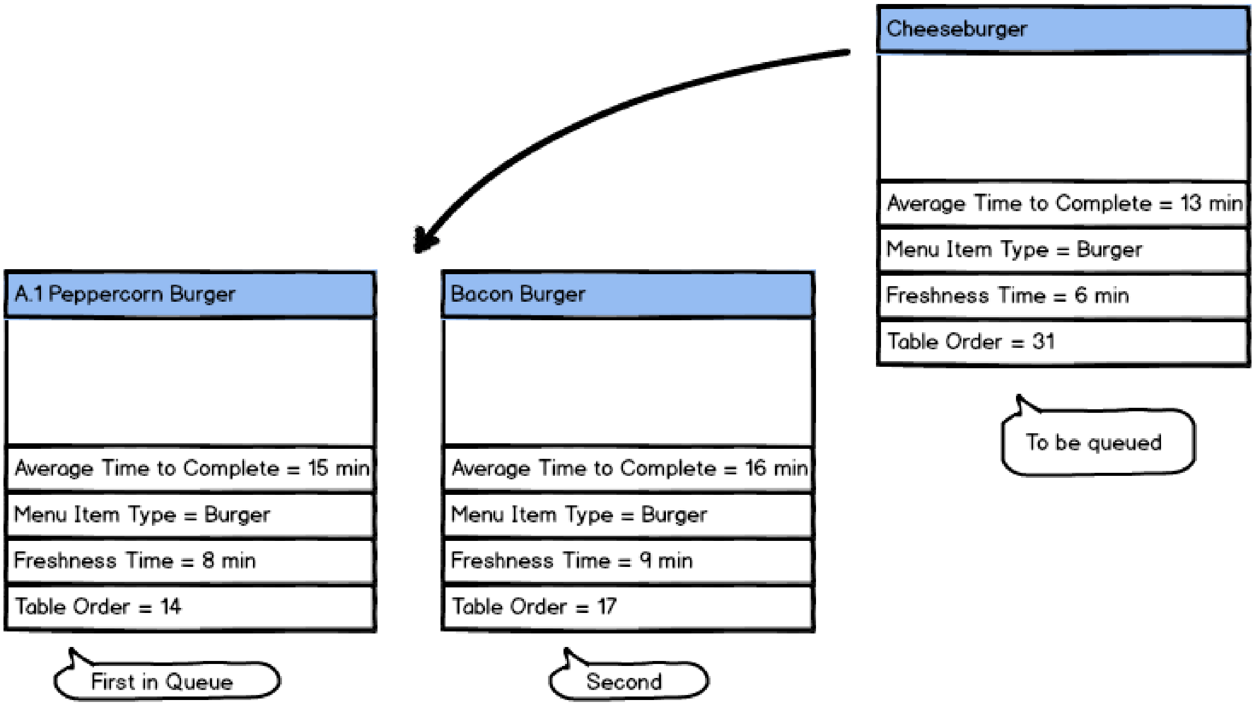


Figure 7: The Cheeseburger is being queued ahead of the Bacon Burger because it has a lower average time to complete.

In this situation, we have multiple burgers that already on the queue and we are adding another burger. In this case, we can find the first burger that has an average time that is greater than the cheeseburger's average time to complete. Thus, we when we are scheduling the Cheeseburger, there is more room for other burgers to be scheduled in the future.

In the previous two scenarios we have neglected the fact of what table the burgers are being ordered from. Suppose we have a group of people that are ordering from the same table. In our current scheduling policy, some members of the group will have food that will be cooked far ahead of others in the table. Therefore, some of the food will not be as fresh as others. Even if the food is still kept warm, certain foods rely on freshness for their taste and consequently customer satisfaction.

Therefore, we added a certain freshness factor to the policy whereby dishes cannot be put ahead of other dishes from the same table by this freshness time. Thus dishes cannot be queued earlier than the freshness time away from the rest of the table order.

Here's is an example of when freshness time comes into play:

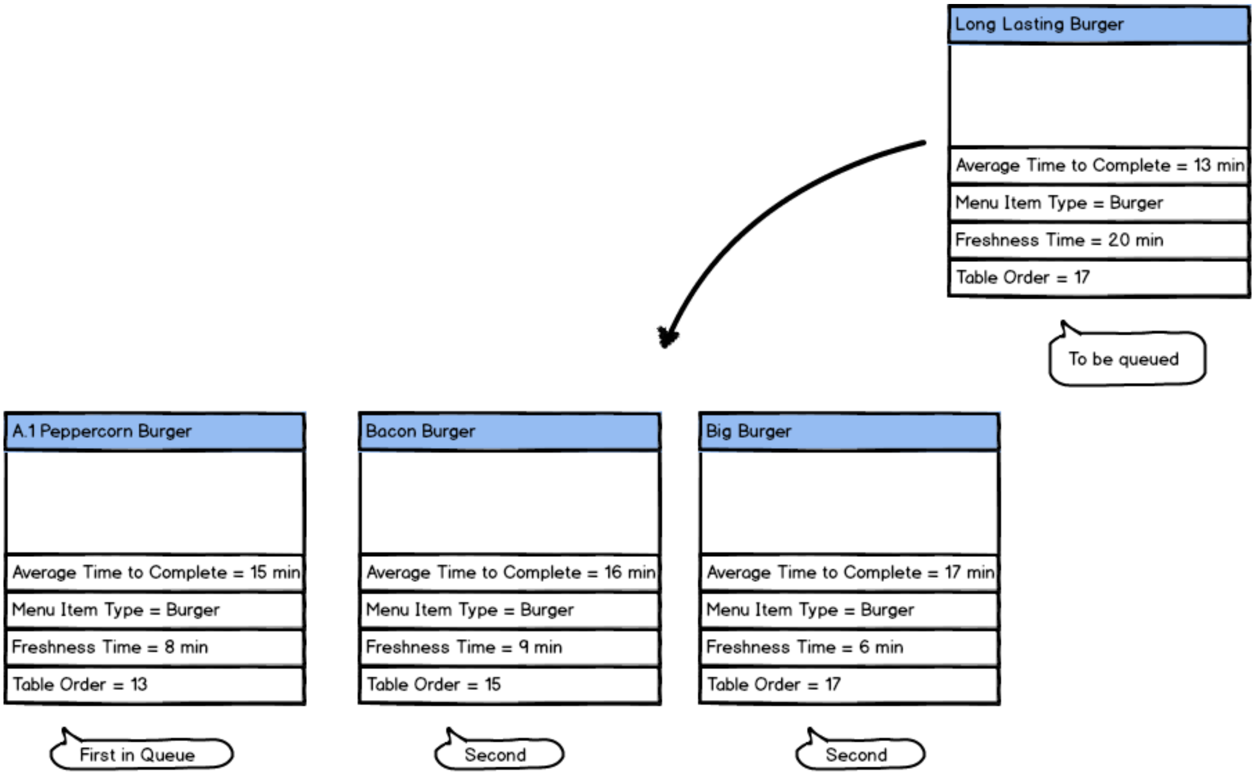


Figure 8: The long lasting burger is being queued with the Bacon burger and not the peppercorn burger because of freshness time constraints.

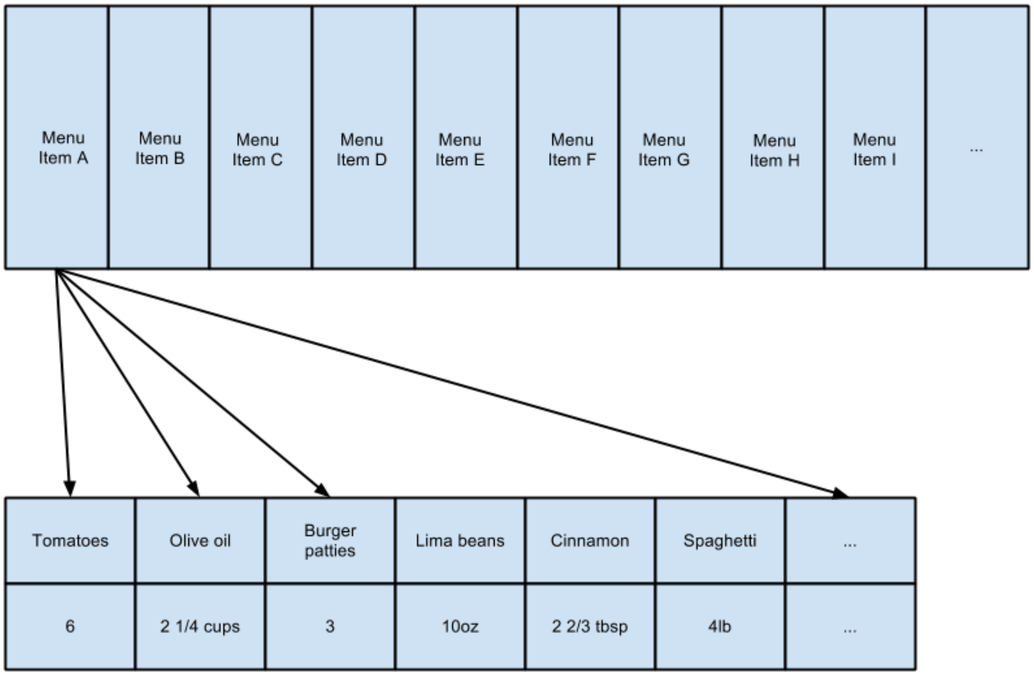
The first thing you may notice is that "Table Order" is another property of the menu item. The table order will be described in a later section, but this attribute is just a reference to the table order that contains this menu item when it was ordered.

What happens in this situation is that the long lasting burger cannot be queued with the peppercorn burger because the time that it would stay out would be larger than its freshness time. Therefore it will be queued with the next best item, which is queuing with the Bacon burger.

The formal algorithm is described in the mathematical model.

### 2.1.6 Shared Ingredient Display

The chef’s interface will show the menu item queue. To further increase the efficiency of the kitchen, our system will determine the total ingredient usage of each ingredient for all menu items currently residing in the queue. By displaying this data directly to the chef and sous-chefs, it will be possible for the kitchen to prepare ingredients and therefore dishes more quickly.



| Tomatoes | Olive oil | Burger patties | Lima beans | Cinnamon | Spaghetti | ... |
|---|---|---|---|---|---|---|
| 6 | 2 1/4 cups | 3 | 10oz | 2 2/3 tbsp | 4lb | ... |

Figure 9: The Chef will be able to see a list of all the ingredients needed for the the current dishes in the order queue.

Originally, we intended to show which ingredients were needed for every dish in the queue; however, doing this quickly caused an unacceptable amount of UI clutter. The current design only displays these relationships for the dish that is up next to be cooked. This allows the chef to quickly understand what ingredients must be retrieved immediately, while the aggregate ingredient usage list shows what ingredients will be needed in the medium and long term. This enables the chef and kitchen staff to have a clear view of what must be done now and what can be done later.

### 2.1.7 Queuing of Orders for Waiters

Similar in design to the chef's menu item queue, this queue represents finished menu items grouped by table and ready to be delivered to the diner. However, the items of this queue are table orders instead of menu items; whenever an entire table order becomes finished in the kitchen, that table's order is placed on the table queue. The waiter checks this queue from the floor terminal and delivers the menu items in a first come first serve manner. However, most restaurants won't deliver items to tables if all the items in the table is ready. We decided to provide the option to the customer whether to wait out till all the items are ready, or just deliver them on a first come first serve basis.

Therefore, we designed the queuing for table order as two queues. One is the wait queue and one is the ready queue. The ready queue will hold either tables that are ready, when the customer chooses to deliver items when all of them are ready, or menu items that are ready to be delivered to tables where the customer chooses to deliver items on a first come first serve basis. The wait queue is a structure of tables and each table will hold the menu items that belong to that table.

Each table structure can be seen as this:

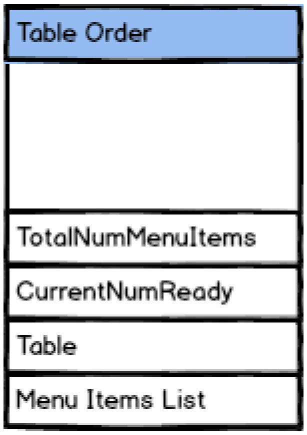


Figure 10: The Table Order structure and its properties that are used in queuing for waiters.

**TotalNumMenuItems:** The total number of menu items associated with this order..

**CurrentNumReady:** The category of menu item that the menu item falls under.

**Table:** This represents the table that the order is coming from. It will just be a number in our case.

**Menu Items List:** This is the list of menu items that belong to this order.

If the customer chooses to deliver items by table, then a table structure is created when he orders as a group. For each menu item that he creates, he adds a reference to it in the Menu Items List. The total number of menu items and the current number that are ready are also set so that one can determine when the table is ready. Table represents

Our Wait queue can then be described as a list of tables as such:

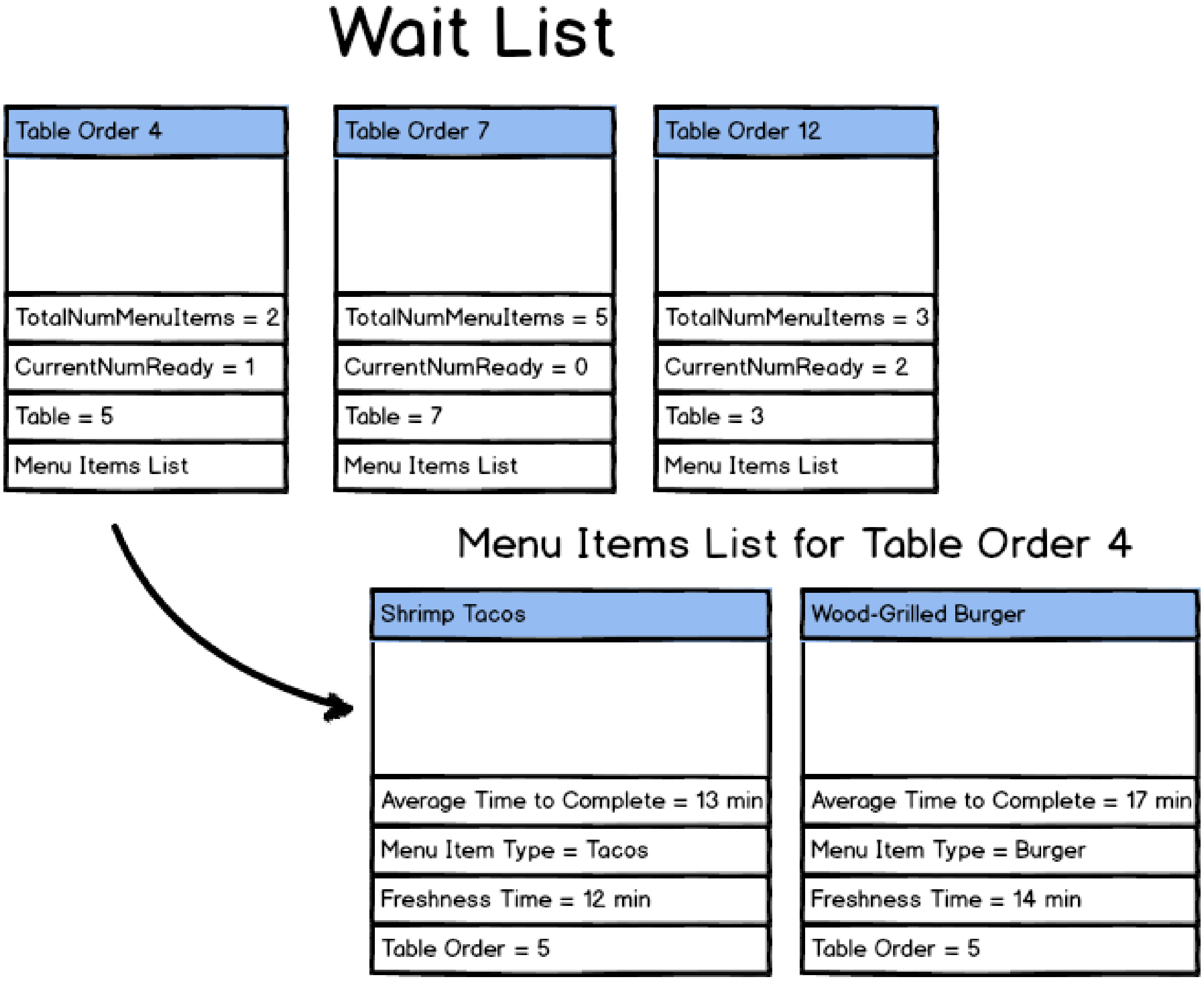
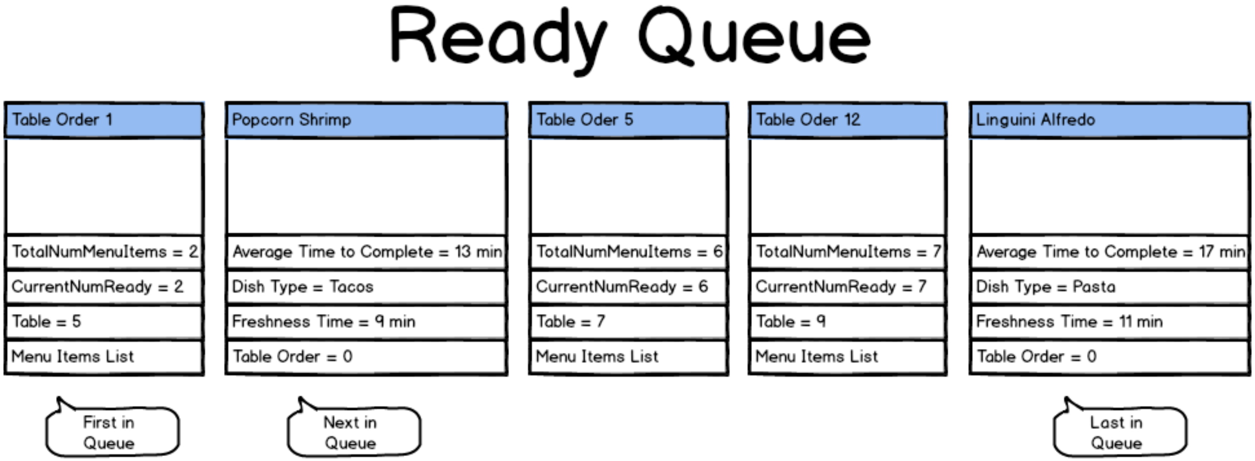


Figure 11: An example of the wait list and an expansion of the menu items list for Table Order 4.

When the customer creates an order, if he selects for the order to be grouped for the table, a table structure will be created and references to the menu Items will be stored in the Menu Item List. Then, when items are done cooking, and are ready to be delivered, they go through each table if the wait list, and if they find themselves in one of the Menu Item lists, then they add the one to the current number ready attribute. When the current number ready is equal to the total number of menu Items, then the table can be queued to the ready queue.

Now the ready queue can contain either tables or individual menu items, again depending on whether the customer chooses to hold out for the group or deliver the item on a first come first serve basis.

The ready queue can be described as follows:



| Table Order 1 | Popcorn Shrimp | Table Oder 5 | Table Oder 12 | Linguini Alfredo |
|---|---|---|---|---|
| TotalNumMenuItems = 2 | Average Time to Complete = 13 min | TotalNumMenuItems = 6 | TotalNumMenuItems = 7 | Average Time to Complete = 17 min |
| CurrentNumReady = 2 | Dish Type = Tacos | CurrentNumReady = 6 | CurrentNumReady = 7 | Dish Type = Pasta |
| Table = 5 | Freshness Time = 9 min | Table = 7 | Table = 9 | Freshness Time = 11 min |
| Menu Items List | Table Order = 0 | Menu Items List | Menu Items List | Table Order = 0 |

Figure 12: The ready queue is based on a first come first serve basis where whatever is added into the queue will get delivered to the appropriate customers. It consists of both Table orders and individual menu items.

The ready queue has a simple scheduling policy, its a first come first serve basis. We can see that it consists of both table structures and individual menu items. This way, depending on whether the customer decided to wait for the group or get his food as soon as its ready, it will be queued accordingly. A table order of zero represents that the menu item does not belong to a table order.

### 2.1.8 Deployment of The System

After designing our solutions to the problems, we were left with the decision of determining how to deploy it in a restaurant environment. The main focus of our deployment was put into making the system simple, high scalability, easy to setup and cost efficiency.

When thinking about how to deploy the system, We noticed that today's restaurant systems are deployed using specific machinery. These machinery are usually built by companies that contributed to the construction of the restaurant. Therefore, they are meant to be static and last the lifetime of the restaurant and consequently they are quite expensive and are rarely expandable.

When we designed our system, we tried to improve on these aspects. We targets a platform for the system that is both cost efficient and easily expandable, namely the android operating system.

Our system will run on numerous android tablets. The manager will have a single tablet that will act as his console. The chefs will have a number of tablets depending on the number of chefs that are at the restaurant. These tablets may be mounted to station so that the chefs may cook and handle the tablet easier. The same goes for the waiters. There will be a tablet at every table so that the customer can interact with the system.

Since Android tablets have become well integrated with society and many people know how to use them well. Therefore, having them instead of built in consoles are incredibly better for a number of reasons.

First, having a tablet system will allow us easily scale the system to fit the needs of the restaurant. A larger restaurant will just need to purchase more tablets than a restaurant of smaller size.
Second, the system is very cost efficient. Android tablets are relatively very cheaper than the custom equipment created to hold the system . Custom equipment like the consoles currently in restaurants.

Third, the system is highly expandable. When the system is deployed on android tablets. Providing updates to the system becomes simple. For every update that is created for the system, it can be pushed to the tablets.

## 2.2 Enumerated Functional Requirements

We felt it was best to use requirements over user stories since some of our ideas were strictly system based. For instance, it was difficult to describe any queuing policy through a user story since the actual scheduling wasn't trigger.

| Identifier | Requirement | PW |
|---|---|---|
| REQ - 1 | The system shall store a database of ingredients and the following information for each ingredient:<br>• Name<br>• Menu items it is used in<br>• Current stock level<br>• Estimated shelf life | 5 |
| REQ - 2 | The system shall store information on the raw ingredients of the menu items such as the estimated shelf life, and the menu items that the ingredient is used in. | 3 |
| REQ - 3 | The system shall store the following information for each menu item:<br>• Ingredients required<br>• Amount of each ingredient<br>• "Freshness" value representing the maximum time this dish should be allowed to wait after being prepared (used in dish queue scheduling) | 3 |
| REQ - 4 | The system shall predict the usage rate of each ingredient and predict the day that the ingredient is expected to fall below a predefined restocking threshold using historical information gathered from the restaurant. | 4 |
| REQ - 5 | The system shall alert the manager when an ingredient's stock level falls below a certain threshold. | 2 |
| REQ - 6 | When an ingredient's stock level falls below its restock threshold, the system shall prepare a restock order and send it to the manager for verification. When it is verified, the system places the order. | 3 |
| REQ - 7 | The system shall predict inventory usage for the next seven days using the previous seven days and show it to the manager. | 2 |
| REQ - 8 | The system shall provide a prediction to the manager when the restaurant will run out of food or fall below a certain threshold in the future. | 5 |

| | | |
|---|---|---|
| REQ - 9 | The system shall give the customer a choice of delivering menu items all at once (by default) or deliver each item to the table on a first come first serve basis. | 3 |
| REQ - 10 | The system shall queue menu items of the same type together so that chefs can cook them in parallel. However, if the customer wants items to be delivered as a table, then items cannot be queued too far ahead of the rest of the table to maintain freshness. The system shall maintain a log of each table order that was placed/edited | 5 |
| REQ - 11 | The system shall show the sous chefs shared ingredients between menu items on the chef's queue so that the sous chef can prepare ingredients beforehand for upcoming menu items. | 2 |
| REQ - 12 | The system shall predict the wait time for menu items that are on the menu and display that information to the customer. | 5 |
| REQ - 13 | The system shall queue orders on a first come first serve basis for waiters based on table, if the customer chooses for orders to be delivered when all orders belonging to the table are ready or individually if he chooses to deliver orders as soon as they are ready. | 5 |
| REQ - 14 | The system shall rank dishes by rating and popularity and display lists of the most popular and highest rated dishes on the menu. | 3 |
| REQ - 15 | The system shall use a menu system to keep a list of all the menu items offered at the restaurant. | 5 |
| REQ - 16 | The system shall allow the manager to manage the menu items on the menu. | 3 |
| REQ - 17 | The system shall allow the information of the menu item to be viewed by the customer. | 5 |
| REQ - 18 | The system shall allow the manager to add, remove, update, and disable menu items on the menu. The system shall also keep a log of the information that is edited on the database. | 4 |

## 2.3 Enumerated Non-Functional Requirements

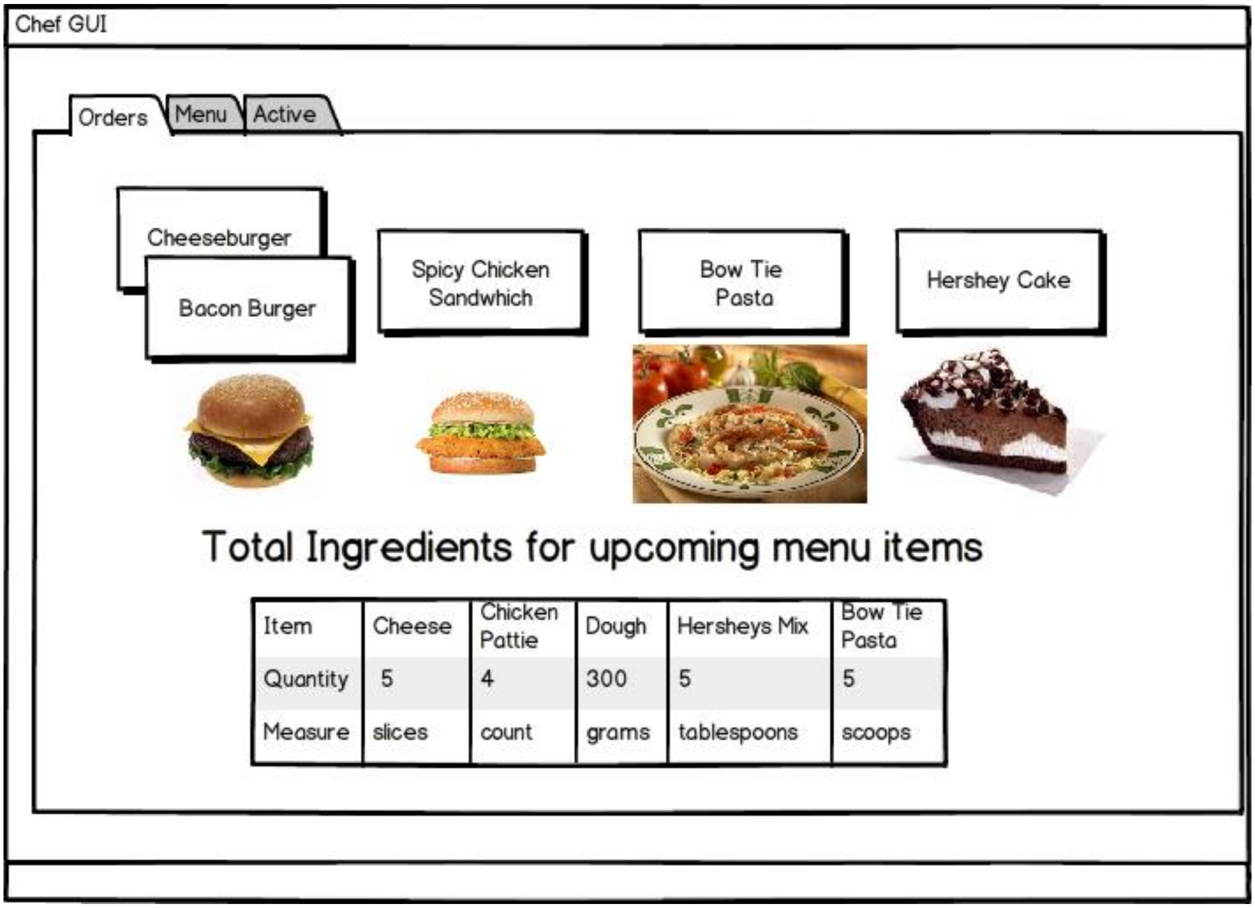
One thing to note is that we do not have any non-functional requirements. The reason for this is that our report details only our ideas rather than a complete system that incorporates requirements that may have been done by another group or trivial requirements such as login, register, display, etc. Since our ideas were completely functional, there were no requirements that were nonfunctional.

## 2.4 On-Screen Appearance Requirements

| Identifier | Requirement | PW |
|---|---|---|
| REQ - 19 | The system shall display the menu and make the menu items selectable to view to the users. | 5 |
| REQ - 20 | The system shall display the average wait time for orders to customer next to the menu items. | 3 |
| REQ - 21 | The system shall display all the ingredients in the inventory so that the manager can view them. | 2 |
| REQ - 22 | The system shall display the ready queue which is list of tables or individual menu items from which users (usually waiters) can select the next item to deliver to the table. | 3 |
| REQ - 23 | The system shall display the order queue to the chef as well as options to select which dish will be prepared. If the dish is unable to be prepared the chef will have the option to disable the menu item. | 4 |
| REQ - 24 | The system shall be able to be able to display an option to the customer allowing him to choose between delivering items on a first come first serve basis or holding out until the table's items are ready. | 1 |
| REQ - 25 | The system shall display the inventory usage for the next seven days to the manager. | 5 |
| REQ - 26 | The system shall be able to show a notification to the manager when it needs to alert the manager in the form of a pop up notification and email. | 5 |
| REQ - 27 | The system shall be able to create a request to the supplier in the form that the supplier specifies. | 3 |

| | | |
|---|---|---|
| REQ - 28 | The system shall be able to send notification to the proper interface. | 4 |

## 2.5 User Interface Mock-Up for On Screen Requirements



Total Ingredients for upcoming menu items

| Item | Cheese | Chicken Pattie | Dough | Hersheys Mix | Bow Tie Pasta |
|---|---|---|---|---|---|
| Quantity | 5 | 4 | 300 | 5 | 5 |
| Measure | slices | count | grams | tablespoons | scoops |

Figure 13: The chef will be able to select the dish of the order queue by touching on the button which indicates the dish has been selected to be prepared. Notice that the first item is actually two items that were queued together.

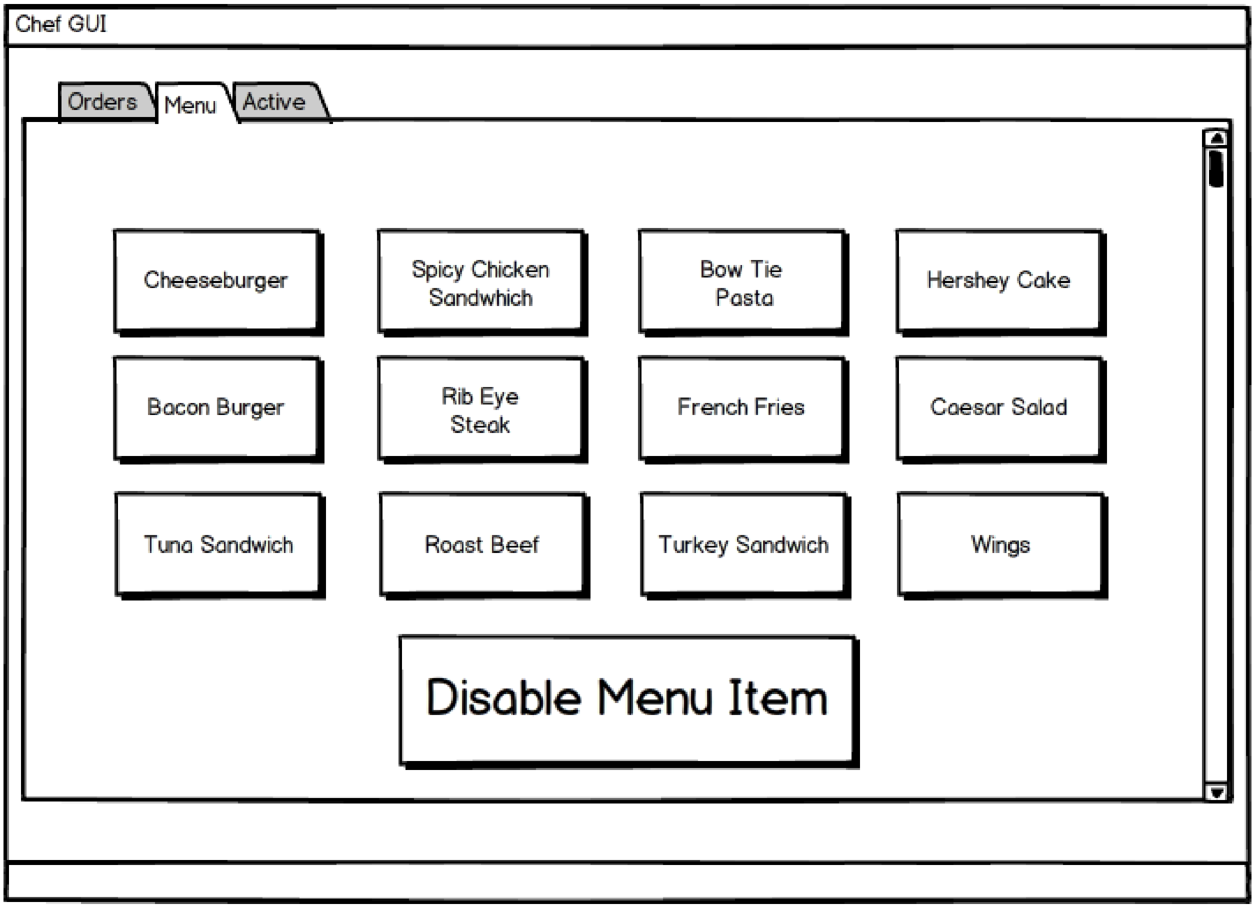


Figure 14: The chef will be able to disable the menu item on the menu if the dish is not able to be prepared by taping on which items he wants to disable ( he can disable more than one at a time) and then tapping the disable menu item button.

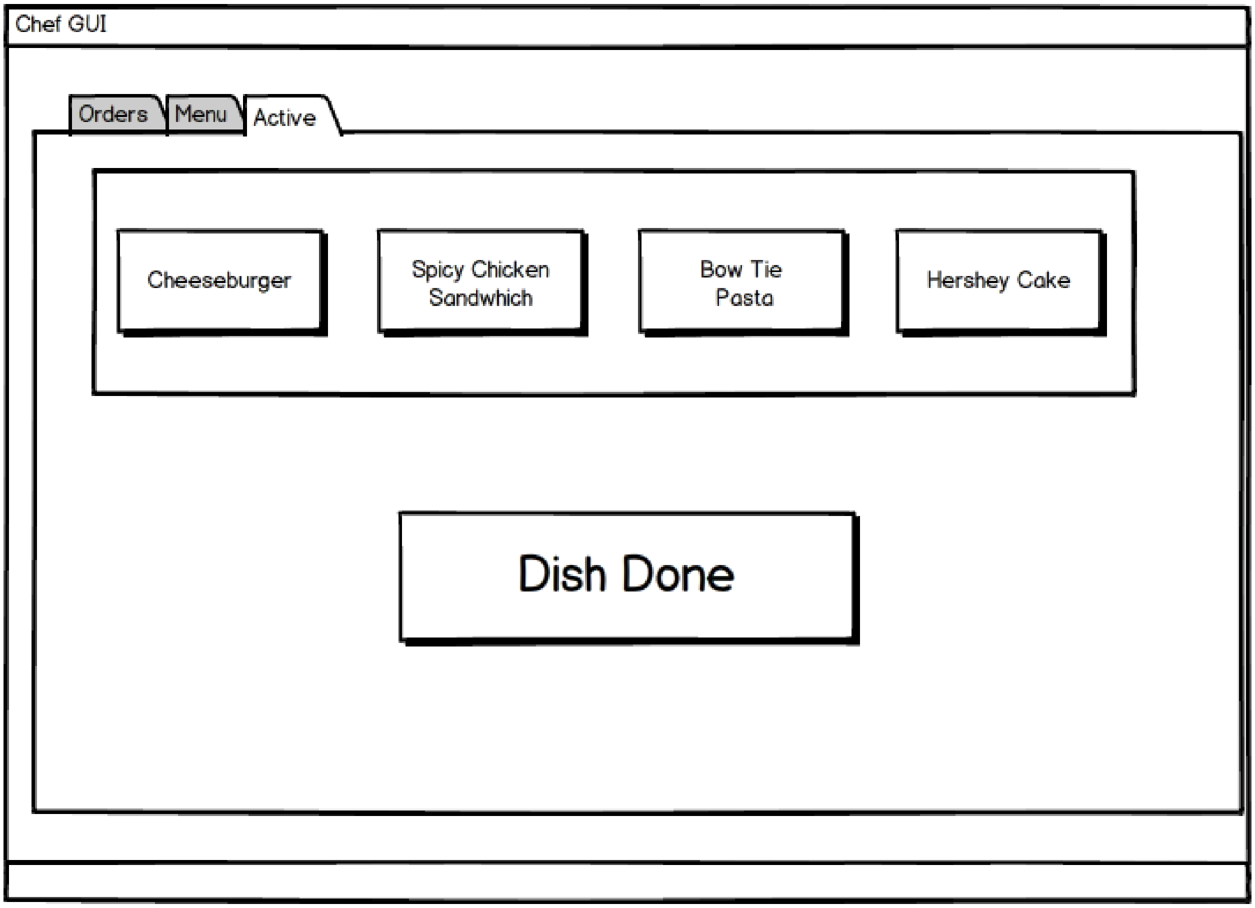
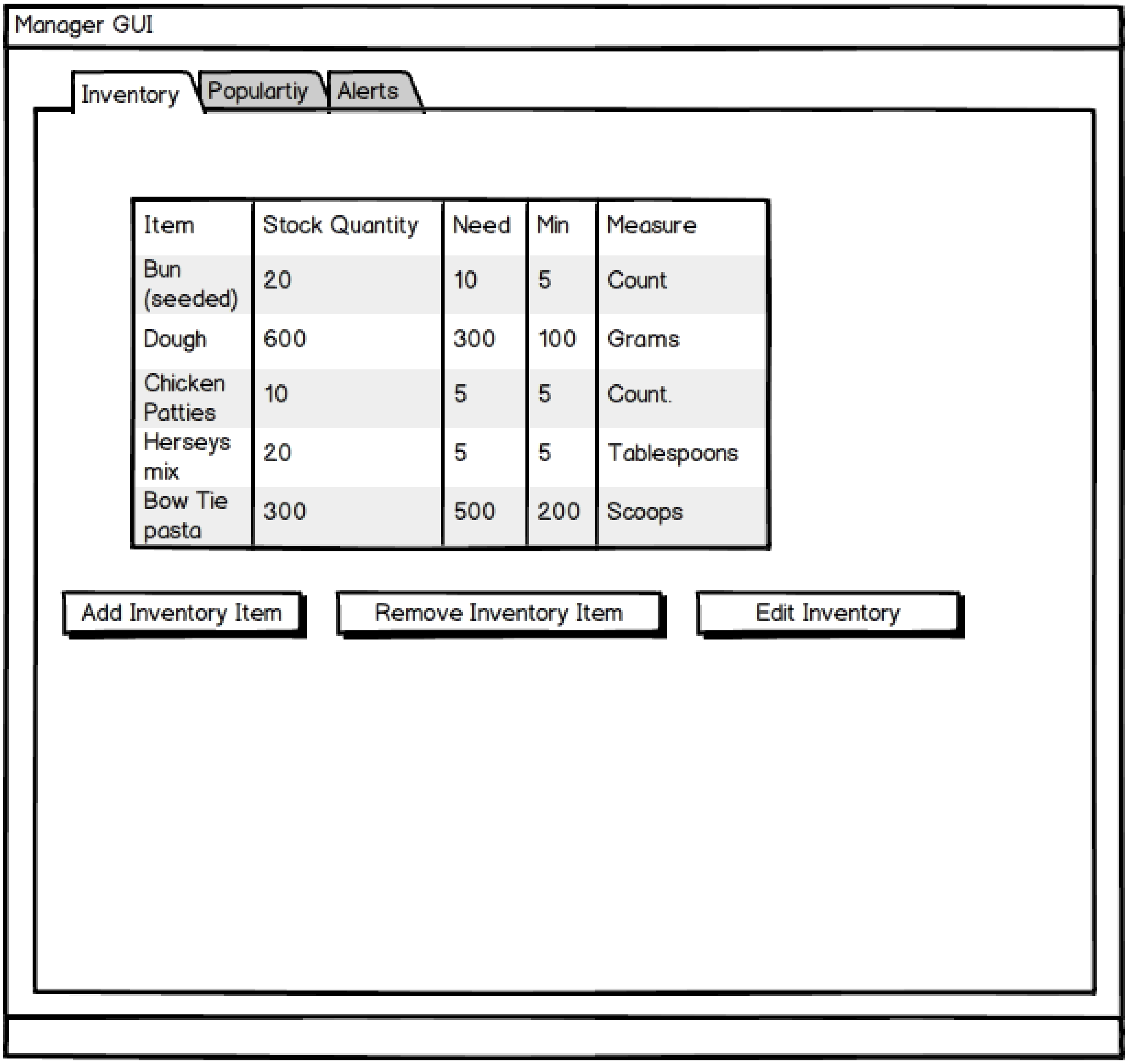


Figure 15: The chef can select which dishes are done being prepared by tapping on the dish and selecting the Dish Done. This will notify the waiter that the dish is ready to be delivered.



| Item | Stock Quantity | Need | Min | Measure |
|---|---|---|---|---|
| Bun (seeded) | 20 | 10 | 5 | Count |
| Dough | 600 | 300 | 100 | Grams |
| Chicken Patties | 10 | 5 | 5 | Count. |
| Herseys mix | 20 | 5 | 5 | Tablespoons |
| Bow Tie pasta | 300 | 500 | 200 | Scoops |

Figure 16: The manager will be able to view a list of the inventory item. The manager will also be able to add, remove, or edit the inventory item.

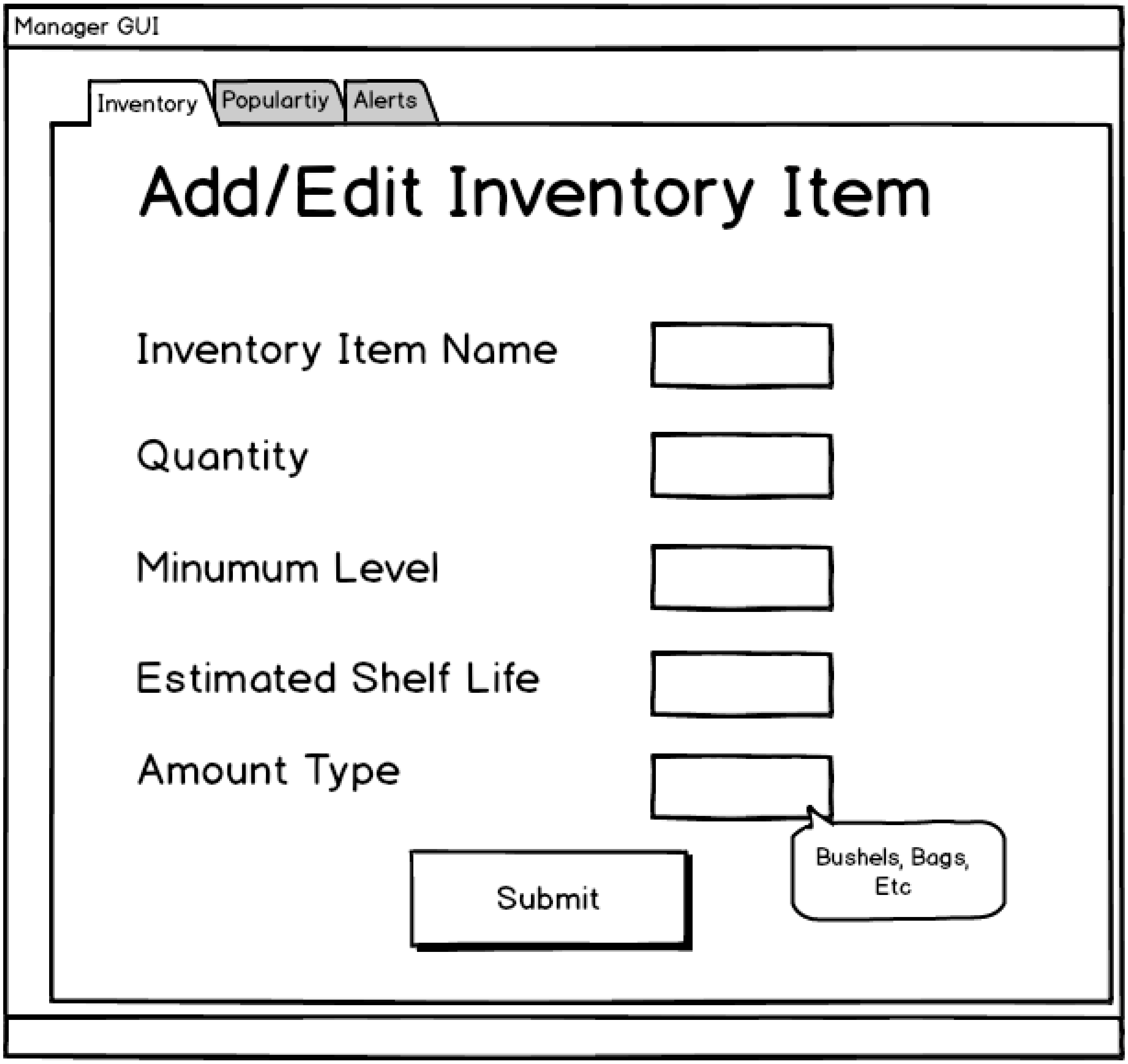
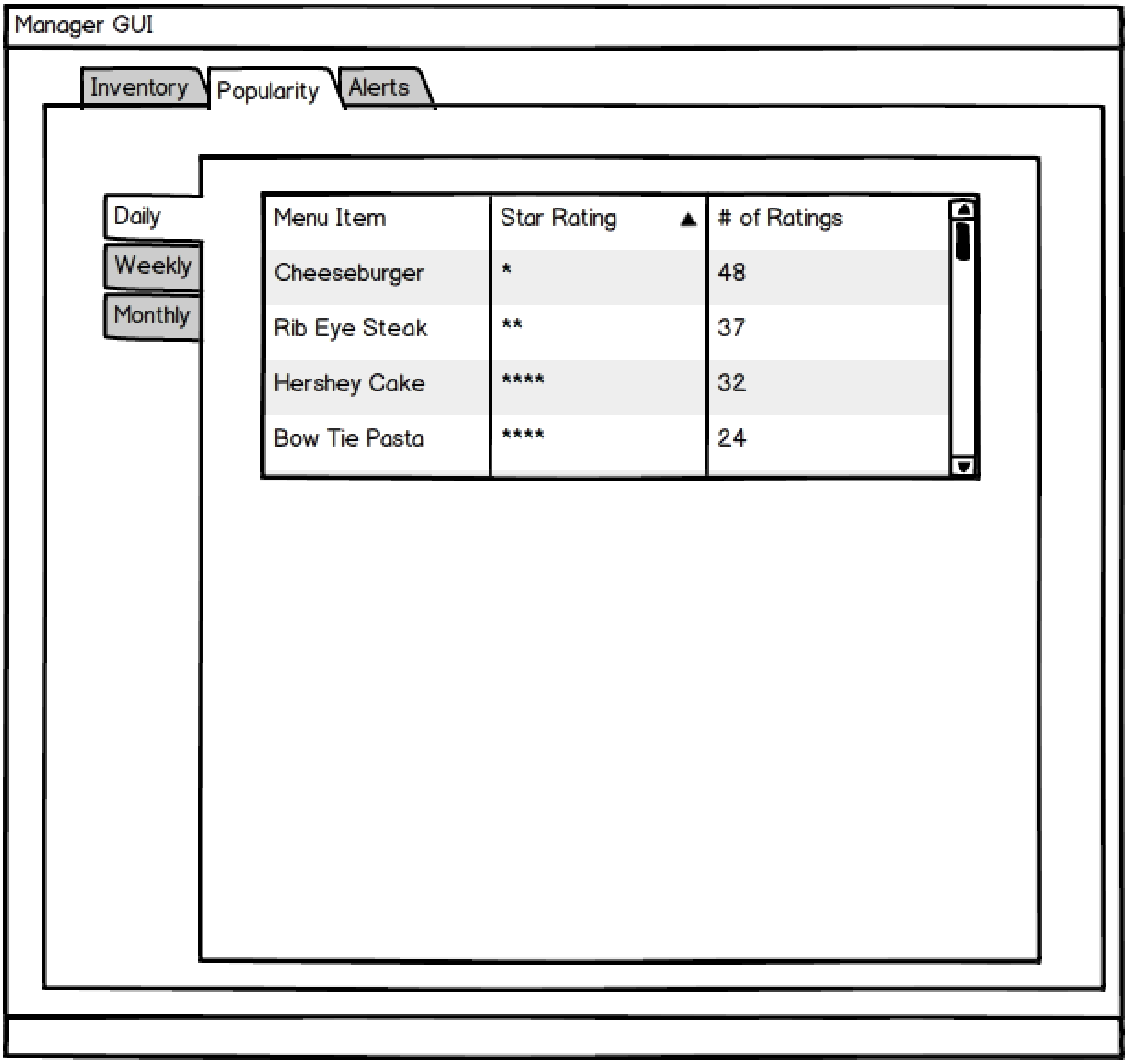
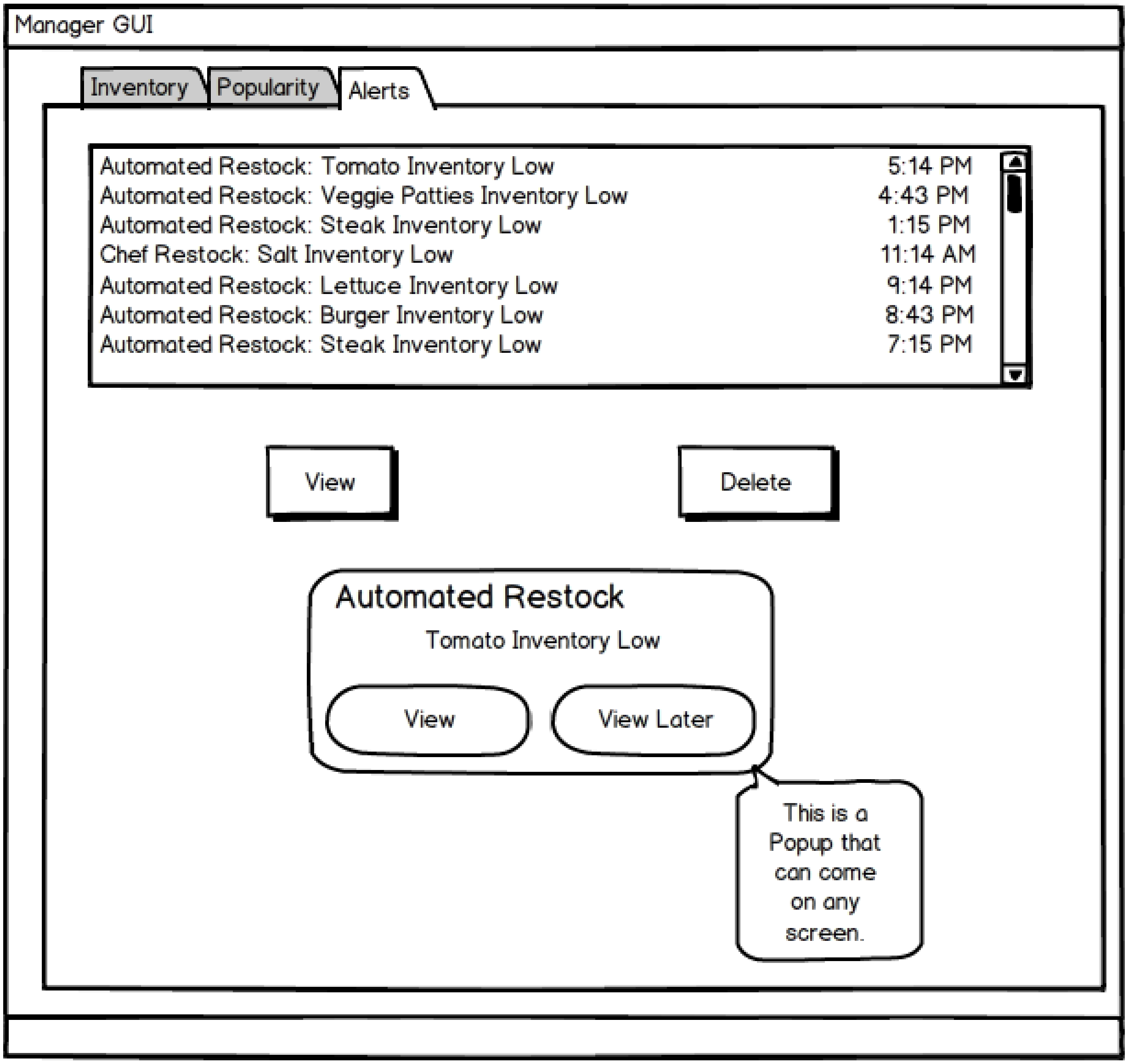


Figure 17: The manager will be able to add and edit inventory items by typing the inventory item name, quantity, etc.



| Menu Item | Star Rating ▲ | # of Ratings |
|---|---|---|
| Cheeseburger | * | 48 |
| Rib Eye Steak | ** | 37 |
| Hershey Cake | **** | 32 |
| Bow Tie Pasta | **** | 24 |

Figure 18: The manager will be able to view a list of popular menu items in the restaurant according to any time reference.



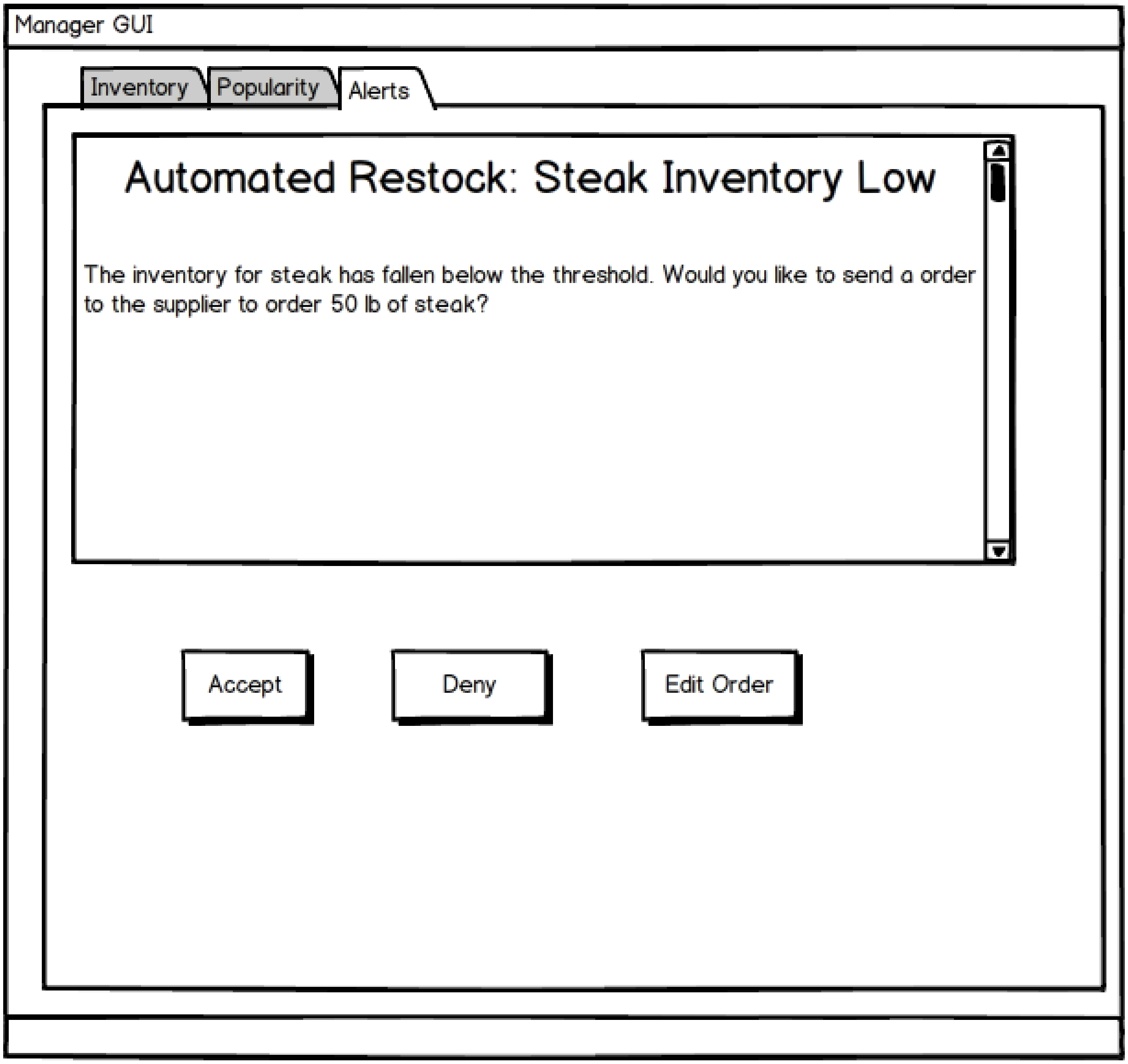
| | |
|---|---|
| Automated Restock: Tomato Inventory Low | 5:14 PM |
| Automated Restock: Veggie Patties Inventory Low | 4:43 PM |
| Automated Restock: Steak Inventory Low | 1:15 PM |
| Chef Restock: Salt Inventory Low | 11:14 AM |
| Automated Restock: Lettuce Inventory Low | 9:14 PM |
| Automated Restock: Burger Inventory Low | 8:43 PM |
| Automated Restock: Steak Inventory Low | 7:15 PM |

Automated Restock

Tomato Inventory Low

This is a Popup that can come on any screen.

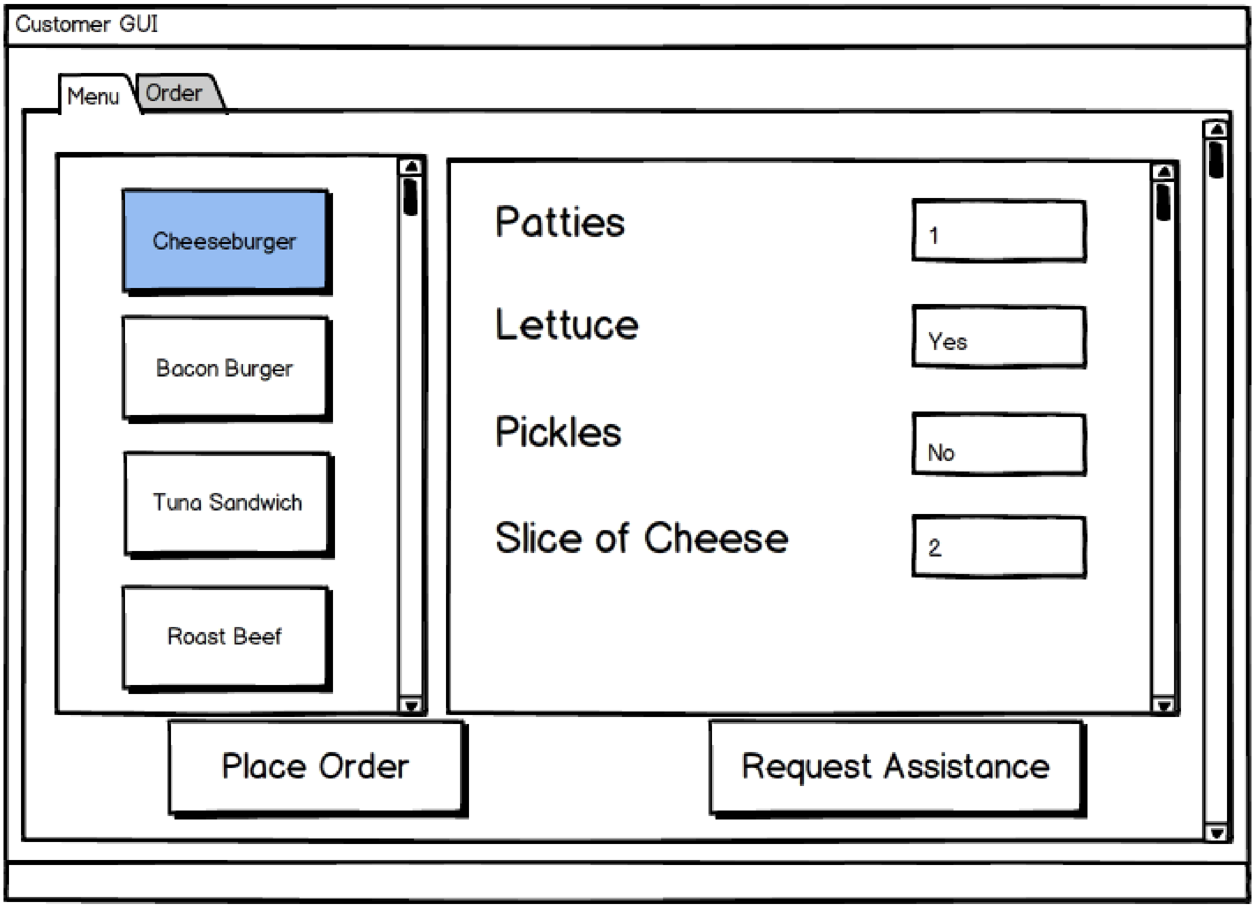
Figure 19: The manager will be able to view a list of all the automated alerts sent by the automated inventory system. The manager will also be alerted in the form of a pop up notification on any screen.



Automated Restock: Steak Inventory Low

The inventory for steak has fallen below the threshold. Would you like to send a order to the supplier to order 50 lb of steak?

Figure 20: The manager will be able to view the alerts and can approve, deny, or edit the request to restock.



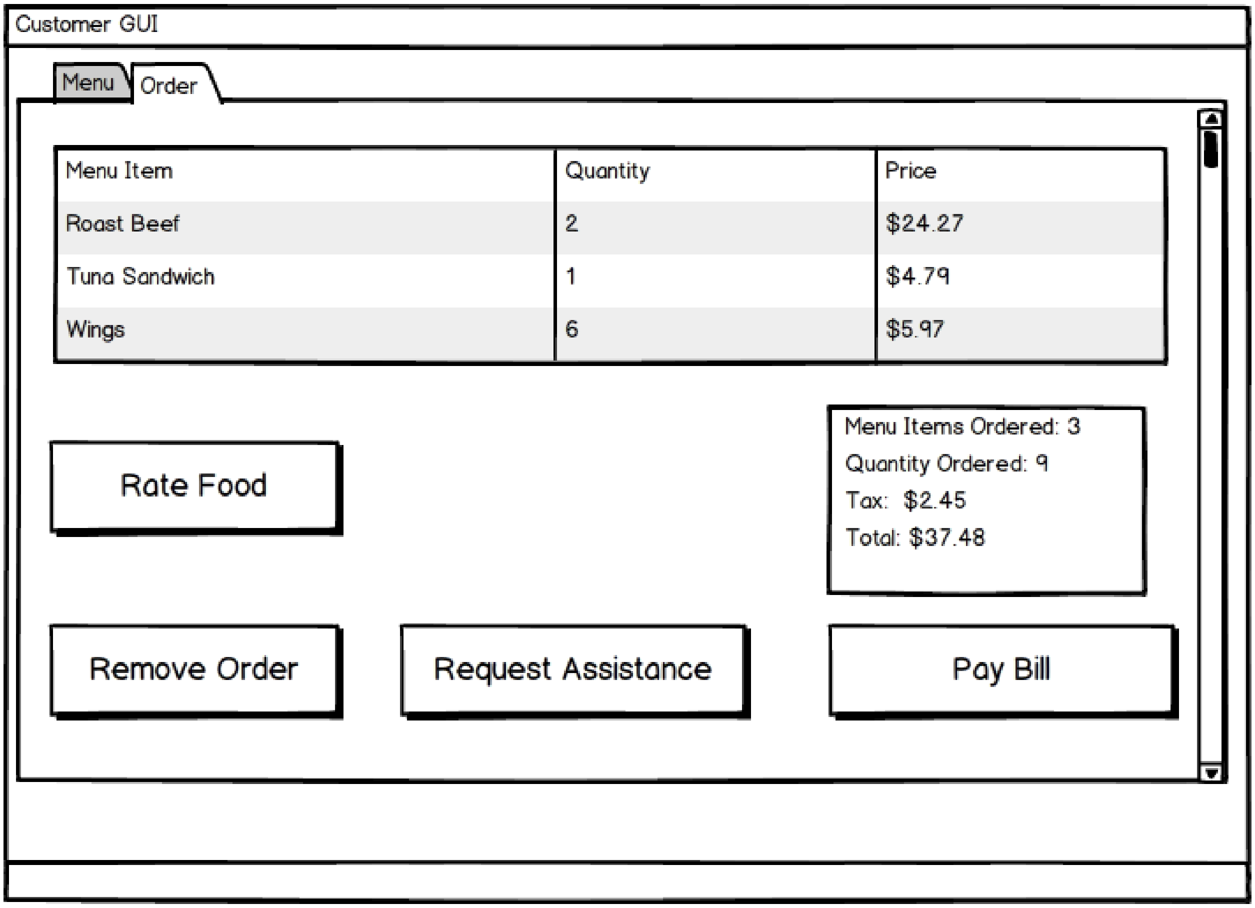
Patties 1

Lettuce Yes

Pickles No

Slice of Cheese 2

Figure 21: The customer will be able to select the menu item and edit the ingredients within it and be able to place the order. The customer will also be able to request assistance of the waiter if needed.



| Menu Item | Quantity | Price |
|---|---|---|
| Roast Beef | 2 | $24.27 |
| Tuna Sandwich | 1 | $4.79 |
| Wings | 6 | $5.97 |

Menu Items Ordered: 3
Quantity Ordered: 9
Tax: $2.45
Total: $37.48

Figure 22: The customer will be able to view the menu item that were ordered along with the price and the total cost so far. The customer will also be able to rate the food, pay the bill, and request assistance by tapping on the appropriate button.

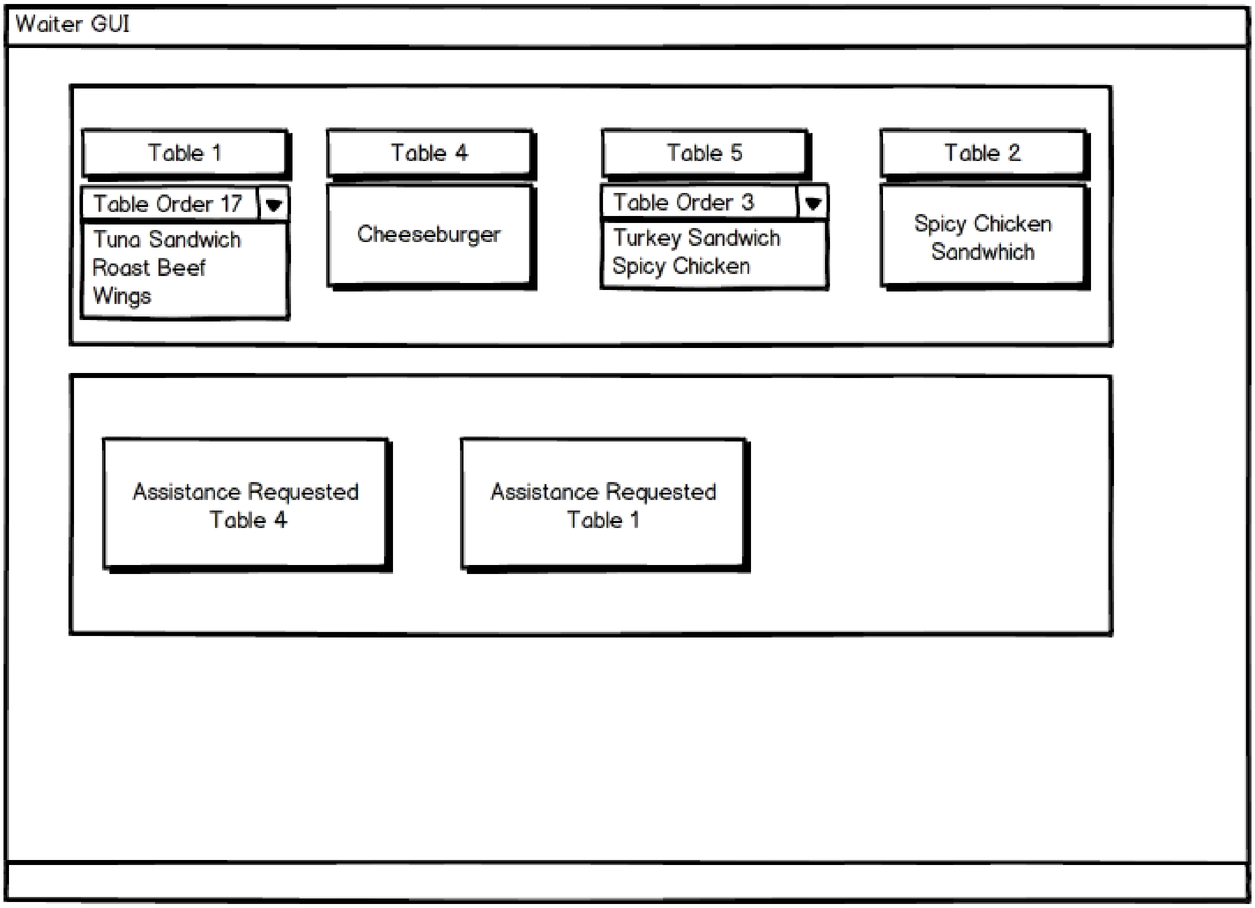


Figure 23: The waiter will be able to see which table requires assistance as well as the dishes that are in the ready queue.

## 2.6. User Effort Estimation

### 2.6.1 Scenario 1: Customer Wishes to Place Order

1. Navigation: Selects the Menu Item (2 Taps)
   A. Customer selects menu item with finger by tapping on it.
   B. After completing the Data Entry as shown below Click Place Order

2. Data Entry: Selects which Ingredients are wanted on the Menu Item (2 or more Tap)
   A. Tap the ingredients that you wish to adjust.
   B. Enter amount or yes/no to adjust number or remove ingredient (max 3 taps)
   C. Tap additional ingredients to add and then tap the add button.

### 2.6.2 Scenario 2: Chef Selecting Dish to Prepare

1. Navigation: Selects the Dish from the order queue to prepare (4 Taps)
    A. Chef taps on the dish that will be prepared and starts preparing it.
    B. After the dish is done being prepared the chef switches to "Active" Tab.
    C. Chef taps on the dish that is prepared and then Dish Done to notify waiter.

### 2.6.3 Scenario 3: Waiter selects order or menu item to deliver

1. Navigation: Selects the next order or menu item on ready queue(1 Tap)
    A. Waiter taps on the menu item and goes to kitchen to pick it up.
    B. Waiter picks up prepared menu item and delivers it to the appropriate table.

### 2.6.4 Scenario 4: Manager wishes to manually add inventory item

1. Navigation: Selects to Add inventory item in the inventory.
    A. Manager taps on the inventory tab.
    B. Manager Selects "Add Inventory Item"
2. Data Entry: Enters appropriate information in the fill in boxes.
    A. Manager enters the Inventory Item Name.
    B. Manager enters the Quantity.
    C. Manager enters minimum level (Threshold).
    D. Manager enters estimated shelf life (Expiration).
    E. Manager enters Amount Type.

# 3. Functional Requirements Specification

## 3.1 Stakeholders

## End Users:

### Restaurant Employees

These are the end users who hold the major interest in the system, as they expect to use it to simplify their life and improving the time efficiency of the restaurant.

### Customers

The end user, who will also be using the system and have some interest in the system as it could possibly lessen their wait time and the need for constant interaction for the waiter.

### Manager

The manager is essentially the administrator of the system and like the restaurant employees has a major stake in the success of the system as he is able to make his restaurant more efficient and easier to manage.

### The Software Team

The software team is the group responsible for the design, implementation and manufacturing of the software and hold the highest interest in the success of the system. In this case, our group is the software team.

## 3.2 Actors and Goals:

## Initiating Actors

### Manager

The restaurant owner who is responsible for managing the system.

### Chef

The chef is the one who cooks all the food.

### Waiter

The waiter is the one who attends customers.

### Customer

The customer of the restaurant who places orders.

## Participating Actors

### Timer

The timer responsible for keeping track of time (When making logs or billing) or clocking the time of an order.

### Database

The storage system used to hold the data (usually local).

# 3.3 Use Cases

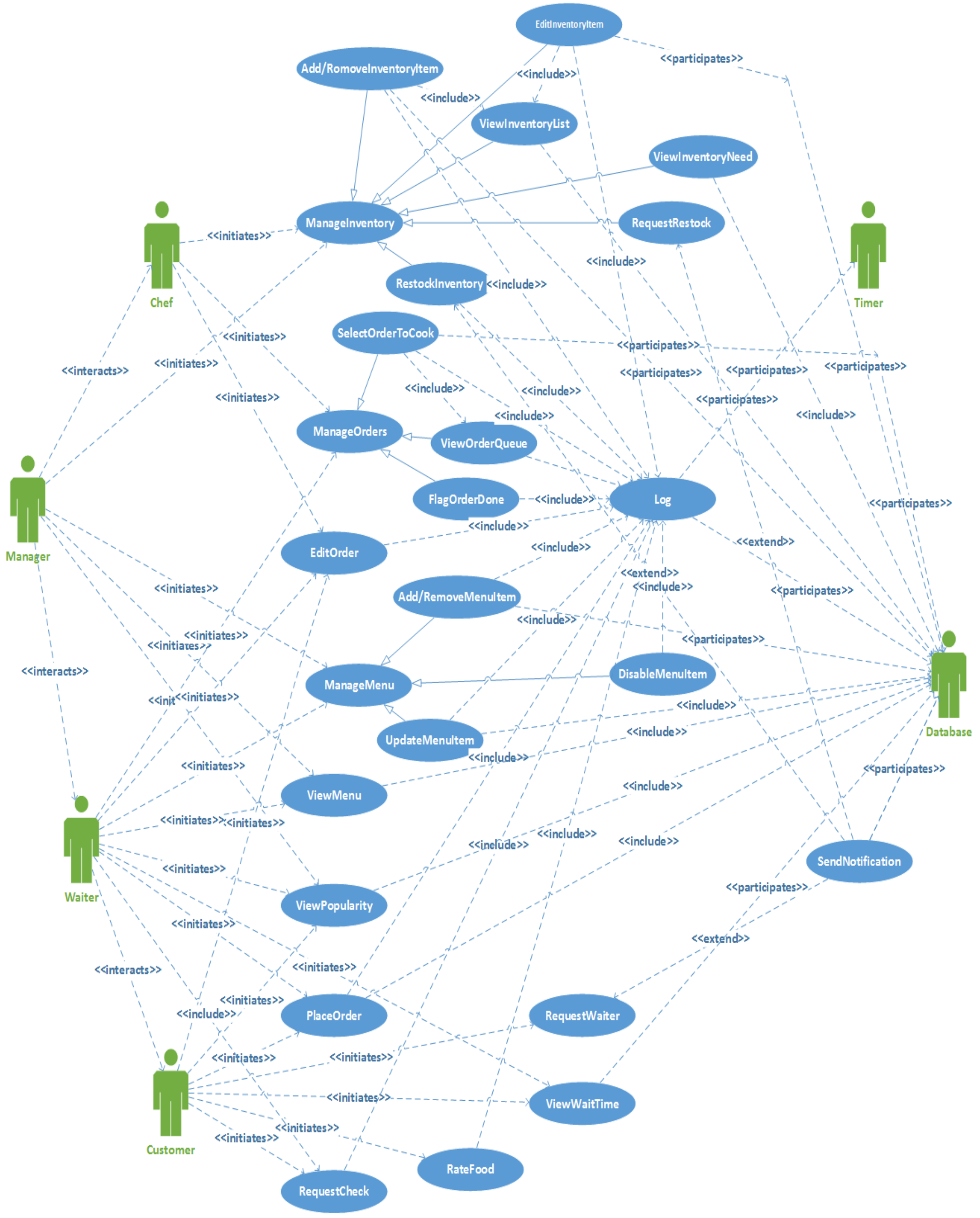
## 3.3.1 Casual Description

| Use Case | Name | Description |
|---|---|---|
| UC - 1 | ManageInventory | Allows the user to manage the inventory. To review more detail on sub use cases refer to UC - 1, 2, 3/4, 5, 6, 7, 27. |
| UC - 2 | ViewInventoryList | Allows the user to view the list of inventoried items along with the each items estimated amount left. (sub use case for UC -1) |
| UC - 3/4 | Add/RemoveInventory Item | Allows the user to add/remove Inventory Item. When item is being added user also has to add specific information concerning the added inventory item which includes, current amount, and quantity measure(what it is measured in), and the shelf life. The updated information is stored as a log( see UC- 20). (sub use case for UC -1) |
| UC - 5 | ViewInventoryNeed | Allows the user to view food trend specific for the past 4 weeks and a list of inventoried Items needed for the coming 7 days. Optional Implementation: system makes use of RequestRestock(UC -6) to automatically make the requests to order inventory needed for the next 7 days. (sub use case for UC -1) |
| UC - 6 | RequestRestock | Allows the system to send a notification (see UC - 26) to the manager to approve a Inventory restock.The updated information is stored as a log( see UC- 20). (sub use case for UC -1) |
| UC - 7 | RestockInventory | Allows user to either  send an order notification for the inventory item to the specified supplier or update information on it manually(see UC - 27).(using sendNotification, reference UC - 26) .A log is created for the information on the order or action. (using log, reference UC - 20)(sub use case for UC -1) |

| | | |
|---|---|---|
| UC - 8 | ManageMenu | Allows user to manage the menu items on the menu. To review more detail on sub use cases refer to UC - 9/10, 11, 12 . |
| UC - 9/10 | Add/RemoveMenuItem | Allows the user to Add/Remove menu items on the menu. When the chef adds the menu item, he is responsible for adding the information about the menu item, which includes: Inventory items (need to make the menu item) along with quantity of each items for the menu item along with the estimated cook/ready time, and the freshness time. The updated information is stored as a log( see UC- 20). (sub use case for UC -8) |
| UC - 11 | UpdateMenuItem | Allows the user to update information on menu items.(reference UC - 9/10 for description on menu item information) The updated information is stored as a log( see UC- 20).  (sub use case for UC -8) |
| UC - 12 | DisableMenuItem | Allows the user to disable menu items that he cannot cook or is unable to and updates the menu information. The updated information is stored as a log( see UC- 20). (sub use case for UC -8) |
| UC - 13 | ViewMenu | Allows the user to view menu containing list of menu items and information on each menu item. (reference UC - 9/10 for description on menu item information) |
| UC - 14 | ManageOrders | Allows the user to manage the Orders that are on the order queue. To review more detail on sub use cases refer to UC - 15, 16, 17. |
| UC - 15 | ViewOrderQueue | Allows the user to view the order queue so that he may select an order to cook/make. (sub use case for UC -14) |
| UC - 16 | SelectOrderToCook | Allows the user to choose an order from the order queue that matches his specialty/skills and removes it from the order queue. The updated information is stored as a log( see UC- 20).  (sub use case for UC -14) |
| UC - 17 | FlagOrderDone | Allows the user to flag the menuItem as ready when the menu item is ready to be served and notifies the waiter.  The updated information is stored as a log( see UC- 20). (sub use case for UC -14) |

| | | |
|---|---|---|
| UC - 18 | PlaceOrder | Allows the user to place orders on the menu items that they want to eat. The order information is stored as a log( see UC- 20). |
| UC - 19 | EditOrder | Allows the user to remove an order or edit its information. The updated information is stored as a log( see UC- 20). |
| UC - 20 | Log | Allows the user/system to log the information of any changes onto the database along with a timestamp. |
| UC - 21 | ViewWaitTime | Allows the user to view the wait time of the menu Items they have ordered or approximate arrival time for a selected a menu item. |
| UC - 22 | RequestWaiter | Allows the user to request the waiter to tend to his or her table. This is done using SendNotification(UC-26). This action is stored as a log( see UC - 20). |
| UC - 23 | RequestCheck | Allows user to request the check after they have finished eating. This is done using SendNotification(UC-26). This action is stored as a log( see UC - 20). |
| UC - 24 | RateFood | Allows user to rate menu items that they have eaten. This action is stored as a log( see UC - 20). |
| UC - 25 | ViewPopularity | Allows the user to view popularity of menu items. |
| UC - 26 | SendNotification | Allows the user/system to send either email notification, with a predefined message body and recipient specific to the type of email, or a notification to their GUI. This action is stored as a log( see UC - 20). |
| UC - 27 | EditInventory | Allows the user to edit information on inventoried item( see UC - 3/4 for more information on editable information). This action is stored as a log( see UC - 20). (sub use case of UC-1). |

## 3.3.2 Use Case Diagrams



### 3.3.3 Traceability Matrix

| | PW | UC1 | UC2 | UC3 | UC4 | UC5 | UC6 | UC7 | UC8 | UC9 | UC10 | UC11 | UC12 | UC13 | UC14 | UC15 | UC16 | UC17 | UC18 | UC19 | UC20 | UC21 | UC22 | UC23 | UC24 | UC25 | UC26 | UC27 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| REQ - 1 | 5 | X | X | X | X | | | X | | | | | | | | | | | | | X | | | | | | | X |
| REQ - 2 | 3 | X | X | | | | | | | | | | | | | | | | | | | | | | | | | |
| REQ - 3 | 3 | | | | | | | | X | | | | | X | | | | | | | | | | | | | | |
| REQ - 4 | 4 | X | | | | | X | | | | | | | | | | | | | | | | | | | | | |
| REQ - 5 | 2 | X | | | | X | X | | | | | | | | | | | | | | | | | | | | | |
| REQ - 6 | 3 | X | | | | X | X | X | | | | | | | | | | | | | | | | | | | | |
| REQ - 7 | 2 | X | | | | | X | | | | | | | | | | | | | | | | | | | | | |
| REQ - 8 | 5 | X | | X | X | X | | | | | | | | | | | | | | | | | | | | | | X |
| REQ - 9 | 3 | | | | | | | | | | | | | | | | | | X | X | X | | | | | | | |
| REQ - 10 | 5 | | | | | | | | | | | | | | X | | X | X | | | X | | | | | | | |
| REQ - 11 | 2 | | | | | | | | | | | | | | X | X | | | | | | | | | | | | |

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| REQ - 12 | 5 | | | | | | | | | | | | | | | | | | | | X | | | | | | | |
| REQ - 13 | 5 | | | | | | | | | | | | | | | | | | X | | X | | | | | | | |
| REQ - 14 | 3 | | | | | | | | | | | | | | | | | | | | | | | | X | X | | |
| REQ - 15 | 5 | | | | | | | | X | | | | | | | | | | | | | | | | | | | |
| REQ - 16 | 3 | | | | | | | | X | X | X | X | | | | | | | | | | | | | | | | |
| REQ - 17 | 5 | | | | | | | | X | | | | | X | | | | | | | | | | | | | | |
| REQ - 18 | 4 | | | | | | | | X | X | X | | X | | | | | | | | | | | | | | | |
| REQ - 19 | 5 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| REQ - 20 | 3 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| REQ - 21 | 2 | X | X | | | X | | | | | | | | | | | | | | | | | | | | | | |
| REQ - 22 | 3 | | | | | | | | | | | | | | X | | | X | | | | | | | | | | |
| REQ - 23 | 4 | | | | | | | | | | | | | | X | | X | | | | | | | | | | | |
| REQ - 24 | 5 | | | | | | | | | | | | | | | | | | X | X | | | | | | | | |
| REQ - 25 | 5 | X | X | | | | | | | | | | | | | | | | | | | | | | | | | |
| REQ - 26 | 5 | | | | | | | | | | | | | | | | | | | | | | | | | X | | |

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| REQ - 27 | 3 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| REQ - 28 | 4 | | | | | | | | | | | | | | | | | | | | | X | X | | | X | |
| MAX PW | | 5 | 5 | 5 | 5 | 5 | 4 | 5 | 5 | 4 | 4 | 3 | 4 | 5 | 5 | 2 | 5 | 5 | 5 | 5 | 5 | 5 | 4 | 4 | 3 | 5 | 4 | 5 |
| TOTAL PW | | 31 | 15 | 10 | 10 | 12 | 11 | 8 | 20 | 7 | 7 | 3 | 4 | 8 | 14 | 2 | 9 | 8 | 13 | 8 | 16 | 5 | 4 | 4 | 3 | 8 | 4 | 10 |

UC 1 > UC 8 > UC 20 > UC 2 > UC 14 > UC 18 > UC 5 > UC 6 > UC 3 = UC 4 = UC 27 > UC 16 > UC 7 = UC 13 = UC 17 = UC 19 > UC = 9 = UC 10 > UC 21 > UC 12 = UC 22 = UC 23 = UC 26 > UC 11 = UC 24 > UC 15

Therefore, we elaborate UC 1, 8, 14, 18(We chose to Ignore UC 2 it is a sub use cases of 1 and will be elaborated in it) as they have the highest priority as they not only enclose most of the sub - use cases while also involving most of the functionality presented for the restaurant. These use cases, ManageInventory, ManageMenu and ManageOrders and PlaceOrders, are the backbone features of our system and so have the highest priority as they take and make up the most significant amount of our system requirements. Although UC-20 (Log) has a high priority weight, we will not be describing this in our fully-dressed or considering it of importance as it is a trivial aspect of our system and only aids in keeping track of the ever-changing database. Also, as this functionality has no influence or interaction with the user, a back-end feature aptly put, but rather a sub use case, or a system use case to be more descriptive, we do not wish to describe it in detail. However, we chose to focus on UC – 21, ViewWaitTime, as it represents one of our systems most essential features that we feel not only is an frontal feature, one to market our product, but also an elaborate use-case that requires description and very involved with the user.

## 3.3.4 Fully Dressed Description and System Sequence Diagrams:

From now on we will focus on these five use cases and elaborate them since they pertain to our ideas.

Use Case UC - 1: ManageInventory
Use Case UC - 8: ManageMenu
Use Case UC - 14: ManageOrders
Use Case UC - 18: PlaceOrder
Use Case UC - 21: ViewWaitTime

Other use cases may be included as sub-use-cases in these five, or will be elaborated at a future time.

### Use Case UC - 1:  ManageInventory

Initiating Actor: Manager, Chef
**Actor's Goal:** To manipulate the information on the current inventoried items.
Participating Actor: Database, Timer
Related Requirements: REQ - 1, 25
**Sub - Use Cases:** UC - 2, 3/4, 5, 6, 7, 27
**Precondition:** The user is either a chef or manager or someone who has privilege to view inventory. The database is up and functional. For UC - 6,7, the send notification (UC - 26) is functioning.
**Postcondition:** The desired manipulation has been made to the database and a log has been stored of the action that took place.

Flow of Events:
**1**. -> **User** selects the Inventory Management Option
**2.** <- **System** shows an interface that displays selectable options which include: View Inventory, Add/Remove Inventory, View Inventory Need,Request Restock, Restock Inventory
**3. User** either

a. -> Selects View Inventory (UC - 2)

1. <- **System** either

**a.** displays a list of inventory items from the database and goes to 6

**b.** is unable to contact database and goes to alt: 4

b. -> Selects Add/Remove Inventory Item (UC - 3/4)

1. <- **System** displays an option of either add or remove

**User** either

a. -> Selects Add option (UC - 3)

1. <- **System** displays a list of information required to be filled(view UC-3/4 for this list)
2. -> **User** fills in all the information and selects done after he completes it.
3. <- **System** either

**a.** enters new Inventory Item into the database and updates the database with the user filled information and and goes to 4

**b.** is unable to contact database and goes to alt: 4

b. -> Selects Remove option (UC - 4)

a. do 3. a. 1.

b. -> **User** selects the inventory item he wants to remove

c. -> **System** requests confirmation of the removal of the inventory item

d. -> **User** confirms action

e. System either

**a.** <- removes new Inventory Item from the database and goes to 4

**b.**<- is unable to contact database and goes to alt: 4

c. -> Selects View Inventory Need (UC - 5)

2. -> **System** either

a. Shows the Inventoried Item usage of the last 4 weeks along with the inventoried items and amount of each inventoried items needed for the next 7 days.

b. is unable to contact database and goes to alt: 4

d. -> Selects Request Restock (UC - 6)

1. do 3. a. 1.

2. -> **user** selects the Inventory Item to restock

3. <- **System** uses SendNotification with information of the inventoried item and a predefined message.

4. include:: Send Notification ( UC - 26) (may be unable to send message and goes to Alt: 4)

5. go to 4.

e. -> Selects Restock Inventory (UC - 7)

1. -> **User** either

a. selects to send an order request

1. ->**System** either

a. if User come from an Request Inventory Alert selects the Inventoried Item on the alert

b. does 3. a. 1.

**1.** -> **User** selects the inventory item to order

2. <- **System** queries how much to order

3. -> **User** selects amount to order

4. <- **System** uses SendNotification with information on amount of inventoried item to order and a predefined message.

5. include:: Send Notification ( UC - 26) (may be unable to send message and goes to Alt: 4)

6. go to 4.

b. selects to update inventory manually

1. ->**System** either

a. if User come from an Request Inventory Alert selects the Inventoried Item on the alert

b. does 3. a. 1.

**1.** -> **User** selects the inventory item to update

2. <- **System** queries how much to update

3. -> **User** selects amount

4. <- **System** either

**a.** <- removes the Inventory Item from the database and goes to 4

**b.**<- is unable to contact database and goes to alt: 4

f. -> Selects Edit Inventory Item (UC - 27)

1. do 3. a 1.

2. -> **User** selects Inventory Item to edit

3. <- **System** either

a. displays list of information of the select Inventory Item from the database

b. is unable to contact database and goes to alt: 4

4.-> **User** updates information and selects done.

5. <- **System** either

a. updates the database with the user filled information and and goes to 4

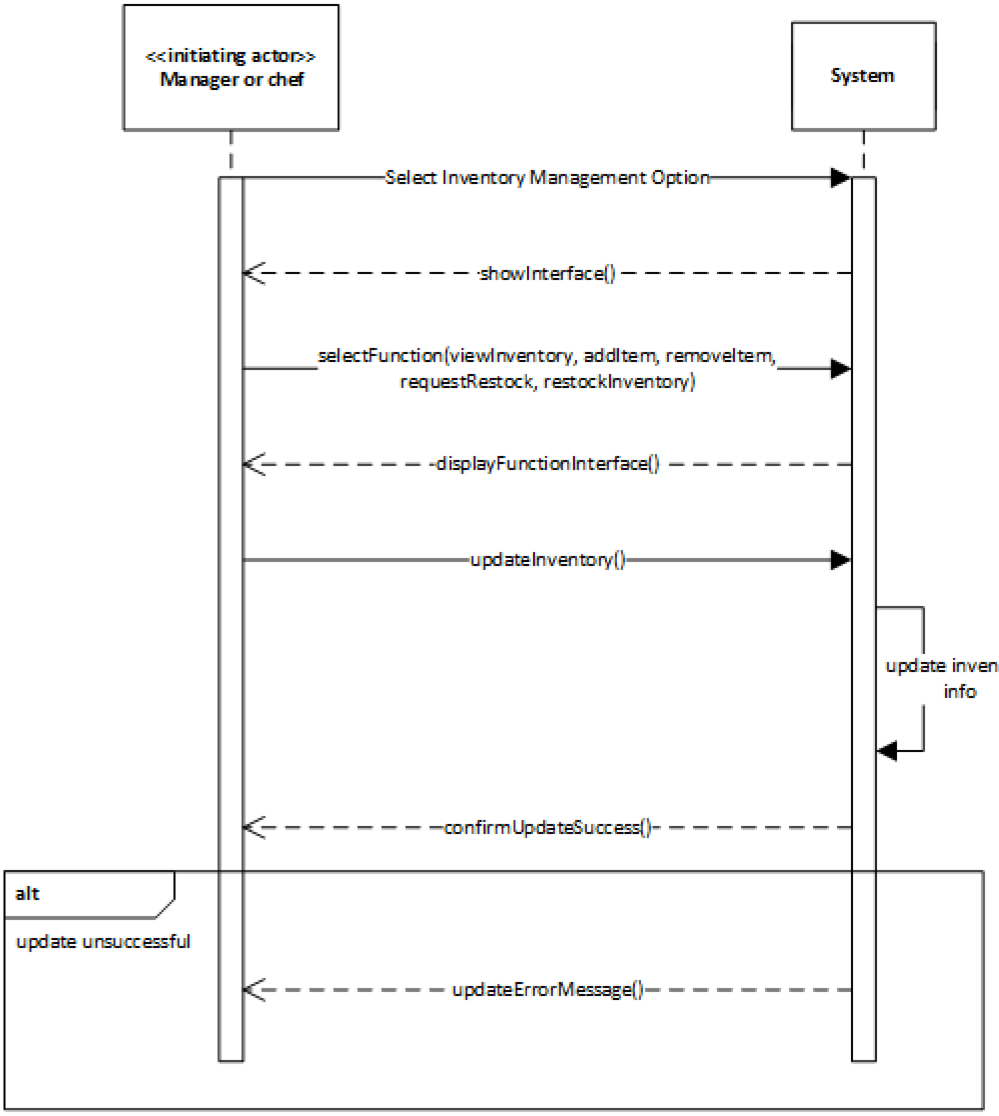
b. is unable to contact database and goes to alt: 4

**4.** <- **System** confirms with a success message.

**5.** Include::log (UC - 20) (Makes a log of any updated information)
**6.** <end>
Alt:
**4.** <- **System** returns an error message
**5.** Include::log (UC - 20)
**6.** <end>

Flow of Events for Main Success Scenario:
any flow of events that pass through all the way to 6 and not through the Alt sequences are success scenarios.
Flow of Events for Fail Scenario:
Any flow of events that lead up to Alt sequences 4-6 will be our fail scenarios.

## Sequence Diagram:



## Use Case UC - 8: ManageMenu

**Initiating Actor:** Manager, Chef, Waiter
**Actor's Goal:** To manipulate the information on the Menu which consists of information on each menu item.
Participating Actor: Database, Timer
Related Requirements: REQ - 3, 15,16,17,18
**Sub - Use Cases:** UC - 9/10, 11,12
**Precondition:** The user is either a chef or waiter or someone who has privilege to view inventory. The database is up and function
**Postcondition:** The desired manipulation has been made to the database and a log has been stored of the action that took place.
Flow of Events:

**1**. -> **User** selects the Menu Management Option
**2.** <- **System** shows an interface that displays selectable options which include: Add/Remove Menu Item, Update Menu Item, Disable Menu Item
**3. User** either

a. -> Selects Add/Remove Menu Item (UC - 9/10)
1. <- **System** displays an option of either add or remove

2. **User** either
a. -> Selects Add option (UC - 9)

a. <- **System** displays a list of information required to be filled(view UC-9/10 for this list)
b. -> **User** fills in all the information and selects done after he completes it.
c. <- **System** either

**a.** enters new Menu Item into the database and updates the database with the user filled information and and goes to 4
**b.** is unable to contact database and goes to alt: 4

b. -> Selects Remove option (UC - 10)

a. include::View Menu (UC - 13)
b. -> **User** selects the Menu item he wants to remove
c. -> **System** requests confirmation of the removal of the Menu item
d. -> **User** confirms action
e. System either

**a.** <- removes the Menu Item from the database and
goes to 4
**b.**<- is unable to contact database and goes to alt: 4

b. -> Selects Update Menu Item (UC - 11)

1. include::View Menu (UC - 13)
2. -> **User** selects Inventory Item to edit
3. <- **System** either
   a. displays list of information of the selected Menu Item from the database

b. is unable to contact database and goes to alt: 4

4.-> **User** updates information and selects done.

5. <- **System** either

a. updates the database with the user filled information and and goes to 4

b. is unable to contact database and goes to alt: 4

c. -> Selects Disable Menu Item (UC - 12)

1. include::View Menu (UC - 13)
2. -> **User** selects the Menu item he wants to disable
3. -> **System** requests confirmation for disabling of the Menu item
4. -> **User** confirms action
5. **System** either

**a.** <- disables the Menu Item and flags it disables in the database and goes to 4

**b.**<- is unable to contact database and goes to alt: 4

**4.** <- **System** confirms with a success message.

**5.** Include::log (UC - 20) (Makes a log of any updated information)

**6.** <end>

Alt:

**4.** <- **System** returns an error message

**5.** Include::log (UC - 20)

**6.** <end>

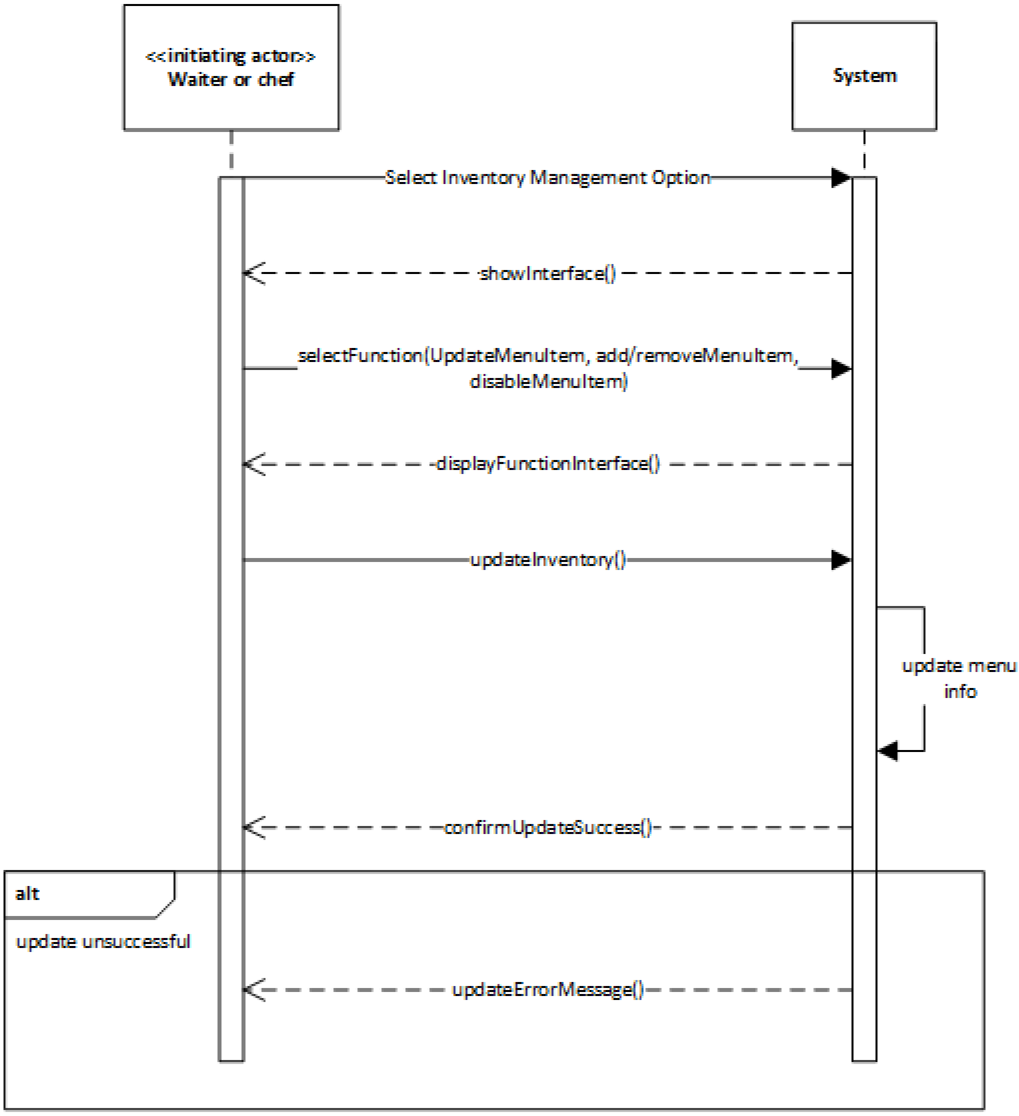
Flow of Events for Main Success Scenario:

Any flow of events that pass through all the way to 6 and not through the Alt sequences are success scenarios.

Flow of Events for Fail Scenario:

Any flow of events that led up to Alt sequences 4-6 will be our fail scenarios.

## Sequence Diagram:



## Use Case UC - 14: ManageOrders

Initiating Actor: Chef, Waiter

**Actor's Goal:** To manipulate information on the orders placed

Participating Actor: Database, Timer

Related Requirements: REQ - 10
Sub - Use Cases:  UC - 15, 16, 17
**Precondition:** The user is either a chef or waiter or someone who has privilege to view inventory.The database is up and running.
**Postcondition:** The desired manipulation has been made to the database and a log has been stored of the action that took place.
Flow of Events
1. ->  **User** selects the Menu Management Option
2. <- **System** displays interface for managing customer orders with options to view the order
queue, select an order to cook, and flag an order as done.
3. -> **User** chooses to either
a. select view order queue (UC -15)
1. <- **System** either
**a.** displays order queue and wait queue from the database and goes to 6
**b.** is unable to contact database and goes to alt: 4

b. select an order to cook (UC - 16)
1. do 3. a. 1.
2. ->**User** select order to cook.
3. **System** either
**a.** <- removes the Menu Item/Items from the Order Queue and puts them
on wait queue and updates the database and goes to 4
**b.** <- is unable to contact database and goes to alt: 4

c. flag an order as done (UC -17)
1. do 3. a. 1.
2. ->**User** select order to flag as done.
3. **System** either
**a.** <- removes the Menu Item/Items from the wait Queue and puts them
on the ready queue and updates the database and goes to 4
**b.** <- is unable to contact database and goes to alt: 4

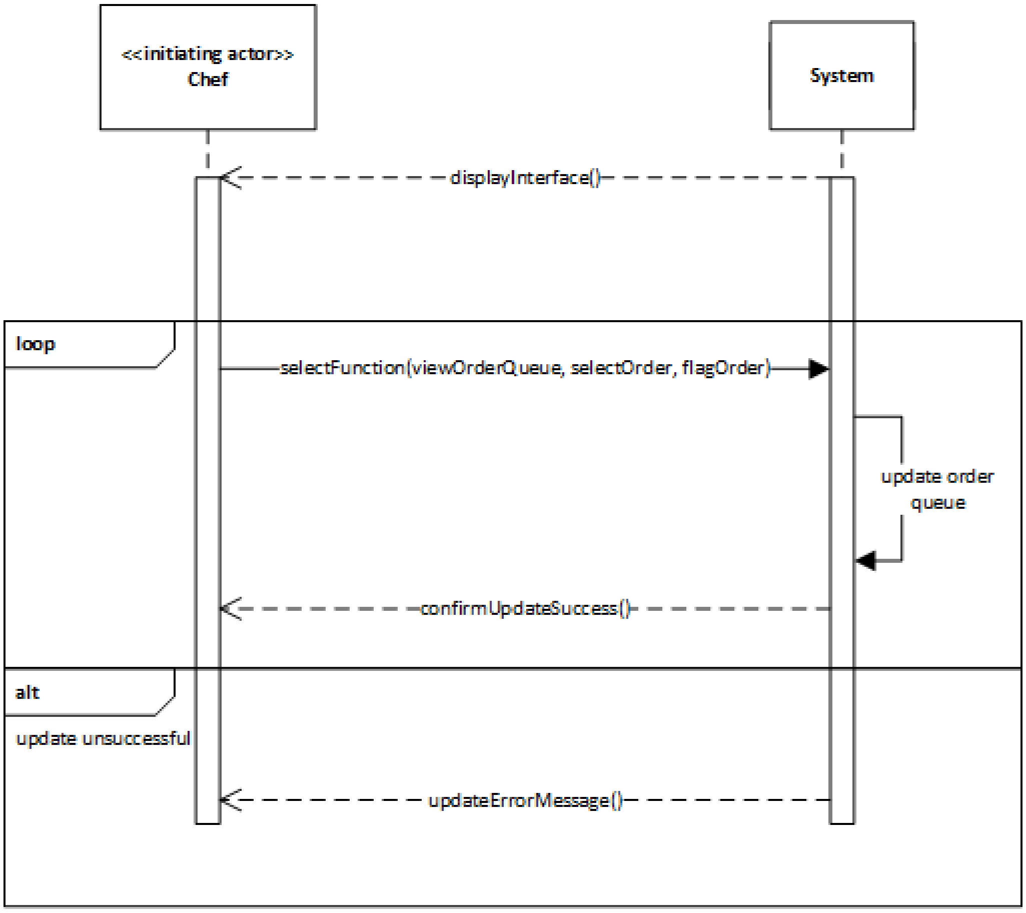
4. <- **System** confirms with a success message.
5. Include::log (UC - 20) (Makes a log of any updated information)
6. <end>

Alt:

4. <- **System** returns an error message
5. Include::log (UC - 20)
6. <end>

Flow of Events for Main Success Scenario:
any flow of events that pass through all the way to 6 and not through the Alt sequences are success scenarios.
Flow of Events for Fail Scenario:
Any flow of events that lead up to Alt sequences 4-6 will be our fail scenarios.

## Sequence Diagram:



## Use Case UC - 18: PlaceOrder

Initiating Actor: Waiter, Customer
**Actor's Goal:** To place an order of menu item/items.
Participating Actor: Database, Timer
Related Requirements: REQ - 13, 24

**Precondition:** The user is either a  or waiter or someone who has privilege to view inventory. The database is up and running. The selected menu item/items are not disabled or unable to be cooked.
**Postcondition:** The selected menu item/items are put on the order queue.
Flow of Events

1. include::ViewMenu(UC - 13)
2. -> **User** selects menu item/items and selects done.
3. <- **System** queries it user wants to make delivery of orders together or individually.
4. -> **User** selects option.
5. **System** either

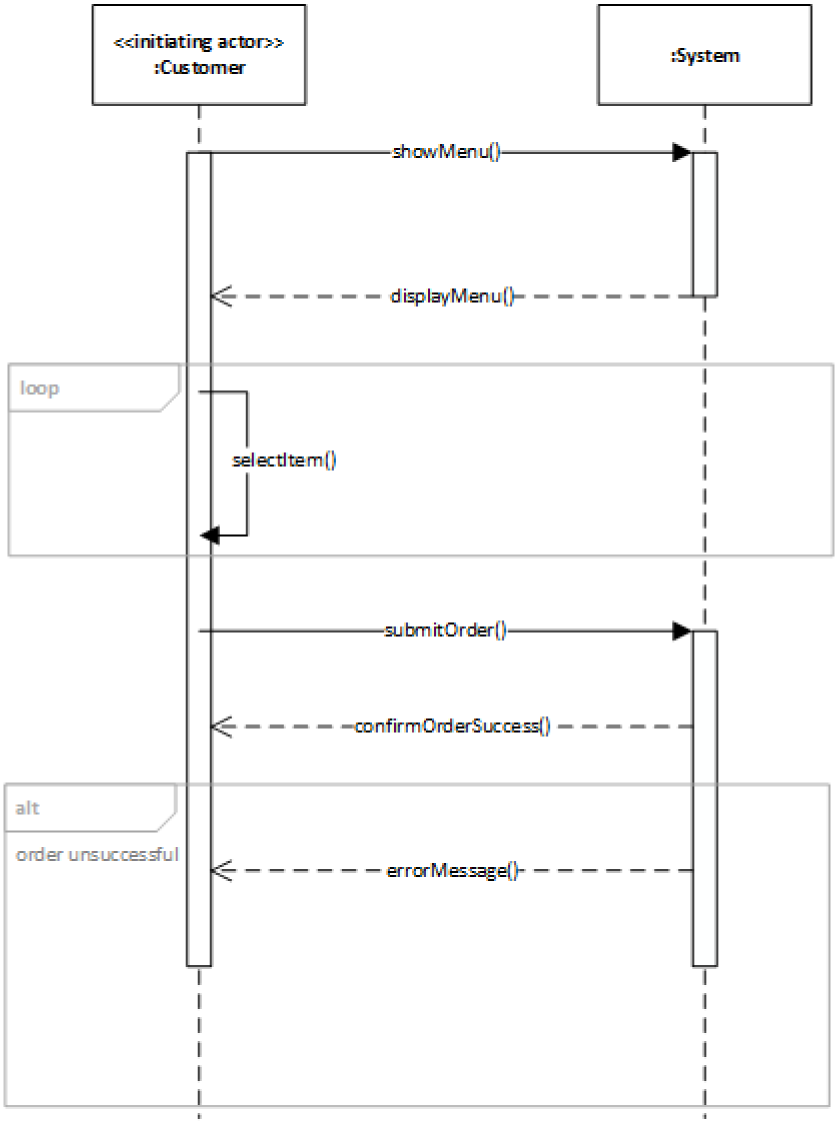
a. <- puts the menu item/items selected on the order queue and updates the database and goes to 6
b. <- is unable to put menu item/items on the order queue as menu item/items has been either disable or is unable to be queued due to low amount of inventory item needed for the the menu item/items and goes to alt: 6
c. <- is unable to contact database and goes to alt: 6

6. <- **System** confirms with a success message.
7. Include::log (UC - 20) (Makes a log of any updated information)
8. <end>

Alt:
6. <- **System** returns an error message
7. Include::log (UC - 20)
8. <end>

Flow of Events for Main Success Scenario:
any flow of events that pass through all the way to 8 and not through the Alt sequences are success scenarios.
Flow of Events for Fail Scenario:
Any flow of events that lead up to Alt sequences 6-8 will be our fail scenarios.

## Sequence Diagram:



## Use Case UC - 21: ViewWaitTime

Initiating Actor: Waiter, Customer
**Actor's Goal:** To view the estimated wait time of either an orders placed (either entire order or specific menu items of the order) or the estimated wait time of a menu item on the menu.
Participating Actor: Database, Timer
Related Requirements: REQ - 12
**Precondition:** The database is up and running.
**Postcondition:** The display shows the wait time of the selected item
Flow of Events :

2. -> **User** selects view wait time and chooses to either
a. View Wait Time of Order
1. -> **System** either
a. shows wait time of item with the most wait time as the time of order if
customer chooses to have all menu items on the order to be delivered
together.
b. shows the wait time of each menu item ordered, if the customer chose
to have the menu items order to be delivered individually.

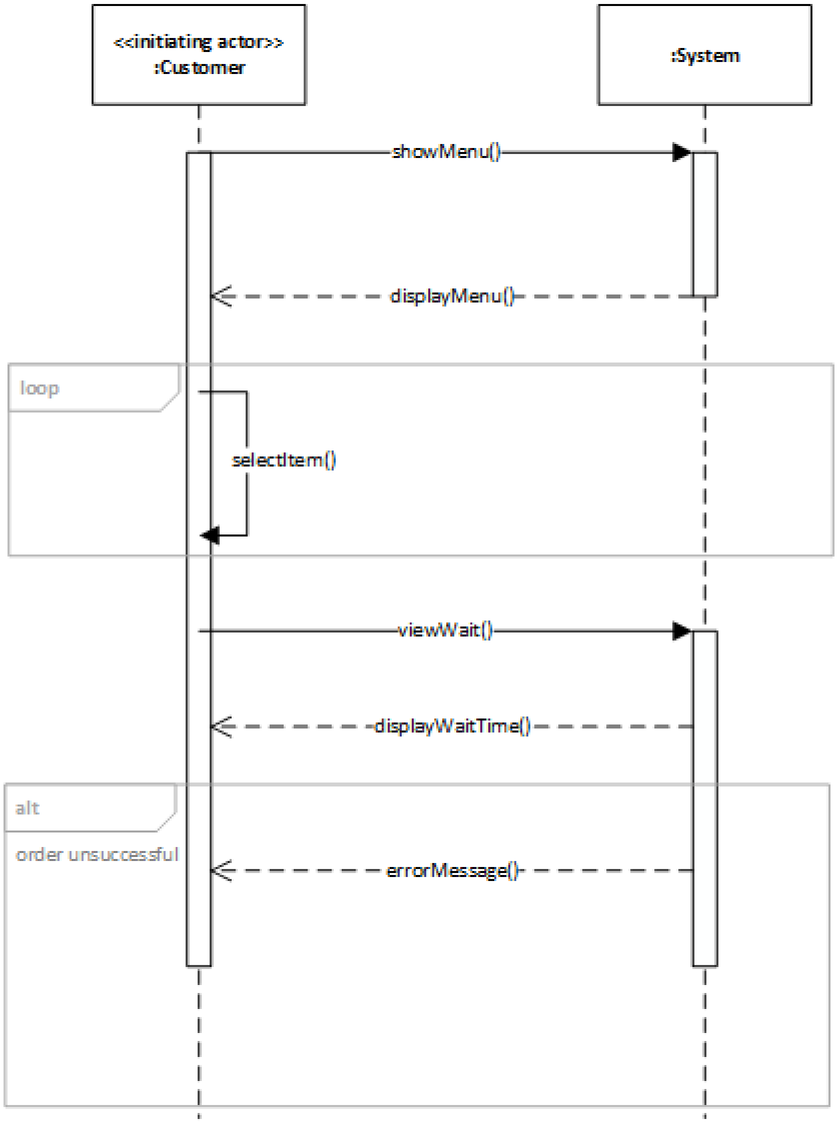
b. View Wait Time of Selected Menu Item
1. -> **System** displays information on select menu item if it was placed on the
order queue and references the database to do so and goes to 3
2. <- is unable to put menu item/items on the order queue as menu item/items
has been either disable or is unable to be queued due to low amount of
inventory item needed for the the menu item/items and goes to alt: 3
3. <- is unable to contact database and goes to alt: 3

3. <- **System** confirms with a success message.
4. Include::log (UC - 20) (Makes a log of any updated information)
5. <end>

Alt:
3. <- **System** returns an error message
4. Include::log (UC - 20)
5. <end>

Flow of Events for Main Success Scenario:
any flow of events that pass through all the way to 3 and not through the Alt sequences are success scenarios.
Flow of Events for Fail Scenario:
Any flow of events that lead up to Alt sequences 3-5 will be our fail scenarios.

## Sequence Diagram:



## 3.3.5 Use Case Points and Effort Estimation

Actor Points:

| Actor | Actor Weight |
|---|---|
| Manager | 3 |
| Chef | 3 |
| Waiter | 3 |
| Customer | 3 |
| Timer | 1 |
| Database | 2 |
| UAW | **15** |

Use Case Points:

| | Use Case Weight |
|---|---|
| UC-1 | 15 |
| UC-2 | 5 |
| UC-3/4 | 10 |
| UC-5 | 5 |
| UC-6 | 10 |
| UC-7 | 10 |
| UC-8 | 15 |
| UC-9/10 | 10 |
| UC-11 | 10 |
| UC-12 | 5 |
| UC-13 | 5 |
| UC-14 | 10 |
| UC-15 | 5 |
| UC-16 | 5 |
| UC-17 | 5 |
| UC-18 | 15 |
| UC-19 | 15 |
| UC-20 | 5 |
| UC-21 | 5 |
| UC-22 | 10 |
| UC-23 | 10 |
| UC-24 | 10 |
| UC-25 | 5 |
| UC-26 | 15 |

| TCF | Technical Complexity Factor Weight | TCF perceived complexity | Complexity Factor |
|---|---|---|---|
| 1 | 2 | 5 | 10 |
| 2 | 1 | 1 | 1 |
| 3 | 1 | 4 | 4 |
| 4 | 1 | 5 | 5 |
| 5 | 1 | 3 | 3 |
| 6 | 0.5 | 4 | 2 |
| 7 | 0.5 | 4 | 2 |
| 8 | 2 | 3 | 6 |
| 9 | 1 | 4 | 4 |
| 10 | 1 | 5 | 5 |
| 11 | 1 | 3 | 3 |
| 12 | 1 | 0 | 0 |
| 13 | 1 | 1 | 1 |
| | | TCF total points | 46 |
| | | TCF | **1.06** |

| | |
|---|---|
| UC-27 | 15 |
| UUCP | **230** |

| ECF | Environmental Complexity Factor weight | ECF perceived impact | CF |
|---|---|---|---|
| 1 | 1.5 | 4 | 6 |
| 2 | 0.5 | 2 | 1 |
| 3 | 1 | 1 | 1 |
| 4 | 0.5 | 3 | 1.5 |
| 5 | 1 | 2 | 2 |
| 6 | 2 | 4 | 8 |
| 7 | -1 | 0 | 0 |
| 8 | -1 | 2 | -2 |
| | | ECF total points | 17.5 |
| | | ECF | **0.875** |

| | |
|---|---|
| Adjusted UCP | 213.325 |
| PF | 28 |
| Duration | **5973.1** |

The values in the above tables were arrived at using the course textbook's methods of effort estimation from section 4.2.1.

# 4. Domain Analysis

## 4.1.1 Concept Definition

| Requirement | Concept | Responsibility | Type |
|---|---|---|---|
| R1 | OrderTaker | Takes the diner's order and sends it to the kitchen. Also capable of updating/editing orders after they are made. | D |
| R2 | Menu | Displays restaurant menu to diner | K |
| R3 | AssistButton | Allows diner to request aid and notifies waiters of the request | D |
| R4 | FoodTimeDisplay | Allows diner to view time until their food arrives | K |

| | | | |
|---|---|---|---|
| R5 | ReadyOrderQueue | Contains knowledge of what orders are done waiting and are ready to be delivered | K |
| R6 | TableStatusView | Displays to the waiter the status of each table in the restaurant: occupied, ready, or needs attention | K |
| R7 | DishQueue | Contains knowledge of dishes that must be made | K |
| R8 | DishCompleteNotifier | Notifies the system that the current dish is complete | D |
| R9 | InventoryDatabase | Contains the total list of ingredients stored in the inventory | K |
| R10 | InventoryNotifier | Notifies manager of the predicted date when an ingredient will run out, when stock for an item falls below a predefined threshold, when an item goes bad, and when restock orders need to be made | D |
| R11 | InventoryChanger | Allows restaurant staff to make changes to the inventory | D |
| R12 | MenuChanger | Allows restaurant staff member to make changes to the menu | D |
| R13 | Logger | Logs order information and restaurant operational information | D |

### 4.1.2 Attribute definitions

| Concept | Attributes | Attribute Description |
|---|---|---|
| Dish | Price | The price of the dish |
| | Ingredients | The ingredients that make up the dish |
| | Popularity | The popularity of the dish |
| Ingredient | Price | The price of the ingredient |

| | | |
|---|---|---|
| | Amount | The amount of the ingredient currently present in the inventory |
| | Type | The type of ingredient: vegetable, meat, dairy, grains, fruits, spices, oils, and other. The other type is a catch-all for items that do not fit into the regular categories. |
| | EstimatedDepletionTime | The estimated time, based on our depletion prediction algorithm, until the ingredient is depleted |
| | EstimatedExpirationTime | The estimated time, based on our expiration prediction algorithm, until the ingredient expires |
| OrderTaker | TableNumber | The number of the table where the order was placed |
| | Dish | The dish that the customer had ordered |
| Menu | Dish | A dish that is currently on the menu |
| AssistButton | TableNumber | The number of the table where assistance was requested |
| FoodTimeDisplay | TableNumber | The number of the table where the request for the arrival time was placed |
| | Dish | The dish the customer requested the arrival time for |
| | Time | The time until the requested dish arrives |
| ReadyOrderQueue | TableNumber | The number of the table where the dish is to be delivered to |
| | Dish | The dish to be delivered |
| TableStatusView | TableNumber | The number of the table displaying its status |
| | TableStatus | The current status of the |

| | | |
|---|---|---|
| | | table: occupied, ready, or needs attention |
| EditOrder | TableNumber | The table number where the updated dish is to be delivered |
| | CurrentDish | The current dish that is to be edited |
| | EditedDish | The updated dish that will be delivered to the customer. If the dish was deleted, this field will indicate it |
| DishQueue | TableNumber | The table number corresponding to where the dish was ordered |
| | Dish | A dish currently present in the queue |
| | DishQueuePosition | The position of the dish in the queue |
| DishCompleteNotifier | TableNumber | The table number corresponding to where the dish was ordered |
| | Dish | The dish that has been completed |
| InventoryDatabase | Ingredient | An ingredient in the inventory database |
| InventoryNotifier | Ingredient | An ingredient in the inventory that has been alerted to the manager by the system |
| | NotificationType | The type of notification sent out by the system to the manager. These notifications can be a predicated date when the ingredient will run out, when stock for an item falls below a predefined threshold, when an item goes bad, and when restock orders need to be made |
| InventoryChanger | Ingredient | The ingredient to be added, |

| | | |
|---|---|---|
| | | changed, or removed from the inventory |
| MenuChanger | Dish | The dish to be changed |

### 4.1.3 Association definitions

| Concept pair | Association description | Association name |
|---|---|---|
| OrderTaker <-> DishQueue | OrderTaker takes orders from tables and sends them to DishQueue for queueing | sends order information |
| OrderTaker <-> Menu | OrderTaker reads menu item information from the menu to construct the object it sends to the DishQueue | reads dish information |
| AssistButton <-> TableStatusView | AssistButton sends the assistance request to TableStatusView for display | sends assistance request |
| FoodTimeDisplay <-> DishQueue | FoodTimeDisplay receives information from DishQueue in order to calculate food wait time | receives dish information |
| ReadyOrderQueue <-> DishCompleteNotifier | ReadyOrderQueue receives finished dishes from DishCompleteNotifier and queues them for delivery to tables | receives finished dish events |
| DishQueue <-> DishCompleteNotifier | DishQueue sends a dish complete event to DishCompleteNotifier | sends finished dish events |
| DishQueue <-> InventoryChanger | When new items are added to DishQueue, DishQueue updates the inventory with the ingredients used in that item | sends inventory update information |
| InventoryDatabase <-> InventoryNotifier | When a warning event happens (restock confirmation, no stock remaining, etc), InventoryDatabase sends the type of event to InventoryNotifier | generates inventory warnings |
| InventoryDatabase <-> | InventoryChanger sends information on | receives |

| | | |
|---|---|---|
| InventoryChanger | what ingredients and fields to update to InventoryDatabase | inventory update information |
| MenuChanger <-> Menu | MenuChanger sends information on what dishes and fields to update to the Menu | sends menu update information |

## 4.1.4 Traceability Matrix

| | C1 | C2 | C3 | C4 | C5 | C6 | C7 | C8 | C9 | C10 | C11 | C12 | C13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| UC1 | | | | | | | | | X | | X | | |
| UC2 | | | | | | | | | X | | | | |
| UC3 | | | | | | | | | X | | X | | |
| UC4 | | | | | | | | | X | | X | | |
| UC5 | | | | | | | | | X | | | | |
| UC6 | | | | | | | | | X | X | | | |
| UC7 | | | | | | | | | X | X | X | | |
| UC8 | | X | | | | | | | | | | X | |
| UC9 | | X | | | | | | | | | | X | |

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| UC10 | | X | | | | | | | | | | X | |
| UC11 | | X | | | | | | | | | | X | |
| UC12 | | X | | | | | | | | | | X | |
| UC13 | | X | | | | | | | | | | | |
| UC14 | | | | | | | X | | | | | | |
| UC15 | | | | | | | X | | | | | | |
| UC16 | | | | | | | X | | | | | | |
| UC17 | | | | | X | | X | X | | | | | |
| UC18 | X | X | | | | | X | | | | | | |
| UC19 | X | X | | | | | X | | | | | | |
| UC20 | | | | | | | | | | | | | X |
| UC21 | | | | X | | | | | | | | | |
| UC22 | | | X | | | | | | | | | | |
| UC23 | | | X | | | | | | | | | | |
| UC24 | | X | | | | | | | | | | | |

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| UC25 | | X | | | | | | | | | | | |
| UC26 | | | | | | | | X | | X | | | |

# 4.2 System Operation Contracts

## 4.2.1 Use Case UC - 1:  ManageInventory

Preconditions:

- IsLoggedIn = True
- IsAuthorized = True
- IsDatabaseOperational = True
- IsNotifierOperational = True

Postconditions:

- DatabaseChange = True
- Database and Inventory have been updated
- The changes made have been saved in the log

## 4.2.2 Use Case UC - 8: MangeMenu

Preconditions:

- IsLoggedIn = True
- IsAuthorized = True
- IsDatabaseOperational = True

Postconditions:

- DatabaseChange = True
- Database and Menu have been updated
- The changes made have been saved in the log

## 4.2.3 Use Case UC - 14: MangeOrders

Preconditions:

IsLoggedIn = True

- IsAuthorized = True
- IsDatabaseOperational = True
- IsQueueEmpty = False

PostConditions:

- DatabaseChange = True
- Database and OrderQueue have been updated

# 5. Mathematical Model

## 5.1 Scheduling Algorithm for Chefs

Since this algorithm is for placing a new menu item into the queue for chefs to cook, it must be run every time a customer places an order.

Our algorithm for scheduling is described as follows:

---

**ItemToBeQueued** - The next item that needs to be added to the queue

**OrderQueue** - The current queue of menu items ready to be cooked by a chef

**CurrentItem** - First Item in Queue, it is not the item that the chef is currently checking. When the chef takes an item from the queue to be cooked, it is removed from the queue.

**Freshness time -** Menu Item "freshness" time or the length of time it can be sitting out.

```
while CurrentItem's Average Time to complete > ItemToBeQueued's Average Time to
Complete

if ItemToBeQueued's  menu item type is the same as the  currentitem's menu item type
        for each menu item at position until end of queue
                if( menu item belongs to same order)
                        length + = menu item's average wait time
                if length < Current Item's freshness factor
                position = CurrentItem's position
Until all menu items in ChefQueue
if( !position )
        position = end of queue.
ItemToBequeued's position = position
```

---

The scheduler will loop through the current queue and find the best place to place the item to be queued. It will check if the item is of the same type as the item to be queued, then it will determine if the average time to complete is greater than that of the item to be queued and also it will check to see if the freshness time rule is kept.  If these things are true, it will store that position and at the very end check if a position was found. If it wasn't found then we know that none of these were met and the item will be added at the end of the queue.

## 5.2 Inventory usage rate estimation and run-out date estimation

Our recursive algorithm estimates ingredient usage for a single ingredient i for day n (today) based on estimated past daily ingredient usage (day n-1, n-2 … n-7) and real-world weekly ingredient usage (week w-1 … w-4). Four weeks was chosen because the window excludes long term seasonal weighting; that is, if a menu item is extremely popular in the winter but not in spring its past popularity won't affect ingredient usage estimates for spring.

Daily ingredient usage will be estimated by adding the usage per menu item of ingredient i for all menu items ordered on that day. Weekly ingredient usage will be real-world data obtained from manual inventory updates by restaurant employees.

In the beginning, there will be no data for past weeks or days. Because weekly information isn't estimated but is based on real-world data, the algorithm will have to take the first few weeks to gather data. At minimum, the algorithm could be run after a single week using only one week's real-world data instead of the average of four weeks. After the first four weeks, the algorithm will be running optimally. Alternatively, the restaurant manager can manually input his own estimate of when an ingredient will run out; until enough real data is gathered, the rate of usage based on this estimate can be used to tune the initial predictions and allow an experienced restaurant manager to guide the system until it has enough data.

First, we determine the relative usage of ingredient i on day n-7 compared to total estimated usage that week, aka the proportion of that ingredient used on the same day of the week last week. That proportion is multiplied by the averaged real-world weekly usage, and yields the estimated ingredient usage for today.

Day n proportional usage: $U(n-7) / (U(n-1) + U(n-2) + U(n-3) + U(n-4) + U(n-5) + U(n-6) + U(n-7))$

Weekly average = $U(w-1) + U(w-2) + U(w-3) + U(w-4)) / 4$

$U_i(n) = [(U(w-1) + U(w-2) + U(w-3) + U(w-4)) / 4] * [U(n-7) / (U(n-1) + U(n-2) + U(n-3) + U(n-4) + U(n-5) + U(n-6) + U(n-7))]$

where U(week) is the actual weekly usage taken from the inventory system's data, and NOT a value returned by the recursive function. This shorthand was used to increase readability.

To take into account holidays and other special occasions, we propose a table of dates and "usage modifiers" which can either multiply an entire day's overall ingredient usage (for nonspecific busy days) or multiply only certain ingredients, such as cranberries on Thanksgiving. With this modification, the final equation is:

$$U_i(n) = M * \{[(U(w-1) + U(w-2) + U(w-3) + U(w-4)) / 4] * [U(n-7) / (U(n-1) + U(n-2) + U(n-3) + U(n-4) + U(n-5) + U(n-6) + U(n-7))]\}$$

## 5.3 Wait time estimation

Basic design for a function WaitTime which returns the wait time for a single item/dish is shown below:

```
WaitTime_D(Dish currentDish)
{
        Time waitTime;
        Time now = currentTime;

        waitTime = currentDish.getArrivalTime() + currentDish.getCookTime() - now;
        return waitTime;
}
```

Finding wait time for an entire table is almost equally as simple: the finish time for a table is equal to the finish time of the dish that is finished last. In simple pseudocode, it could be modeled as such:

```
WaitTime_O(Order currentOrder)
{
Time maxWaitTime = currentTime;
Time temp;
Time now = currentTime;

for this Dish = each dish belonging currentOrder
{
        temp = WaitTime_D(thisDish);
        if temp > maxWaitTime
                maxWaitTime = temp;
}
return now + maxWaitTime;
}
```

## 5.4 Queueing of table orders for waiters

Before we describe the algorithm for queuing, we need to describe how the wait list gets built.

---

Necessary Precondition: Customer chooses to hold menu items until entire table's menu items are ready

```
Create table order with id equivalent to customers table
set total number of menu items
set current number ready to zero
set table = number equivalent to customer's table
create empty MenuItemList

for each menu item in customers order
    add reference to item in MenuItemList

add table order to wait queue
```

---

When the customer creates a composite order and specified that he wants to wait until the the entire order is ready, a corresponding table order will be created. If the customer creates a singular order or specifies that he wants the items on a first come first serve basis, then a table order will not be created. Finally this table order is added to the wait queue. This is important for how the queuing algorithm works.

Now we can describe our final algorithm for queueing table orders for waiters.

---

**ReadyMenuItem** = menu item that has been cooked and needs to be queued.
**Wait List =** The current wait list of table orders
**ready queue** = the current ready queue of either table order or menu items that are ready to be delivered.

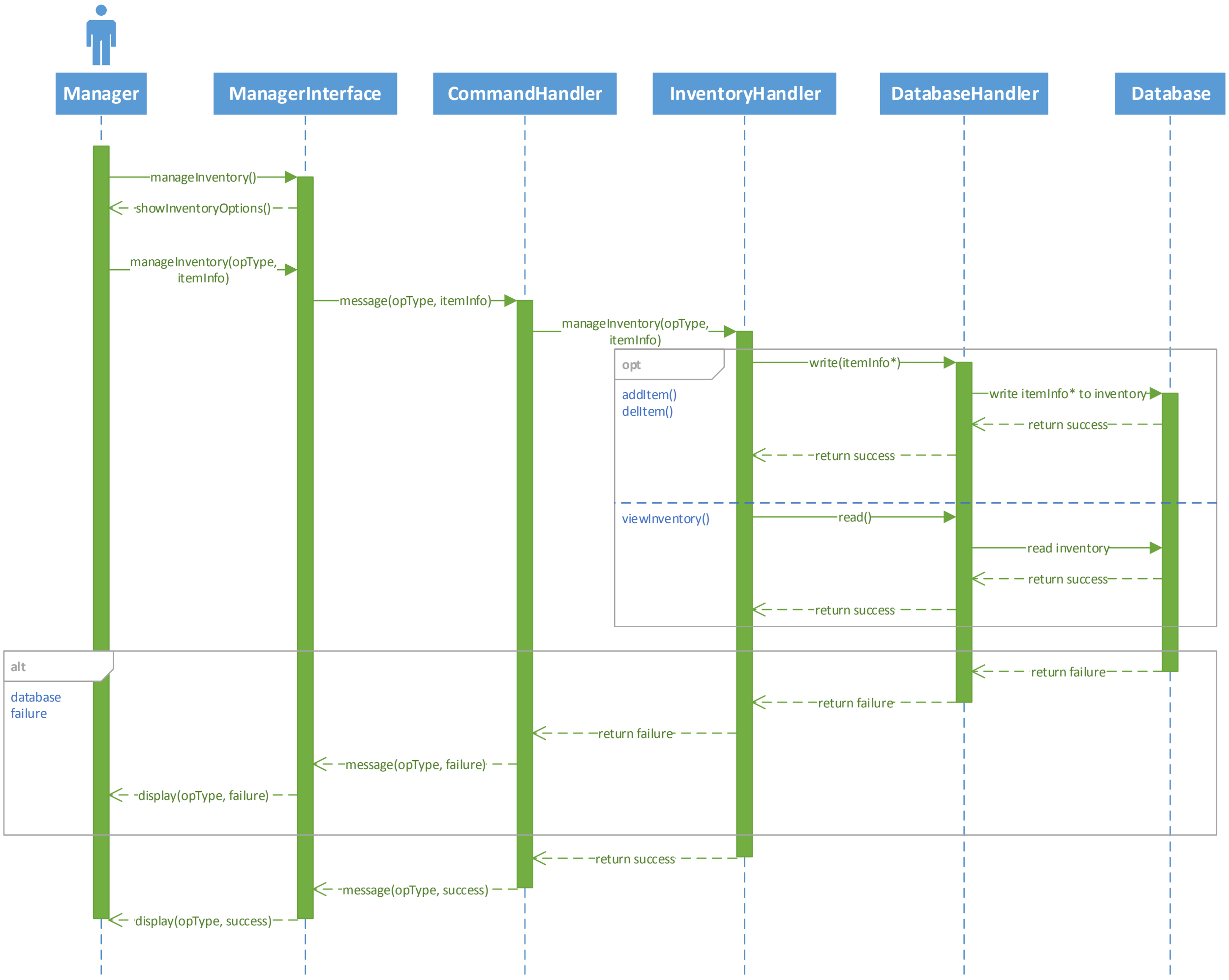
```
boolean foundTable = false
for each table order in wait list
    for each menu item in table order's menu item list
        if readyMenuItem == menu item
            table order.currentNumReady++
            if table order.currentNumReady == table order.totalNumItems
            move table order to ready queue
            foundTable = true


if ( ! foundTable )
add readyMenuitem to Ready Queue
```

---

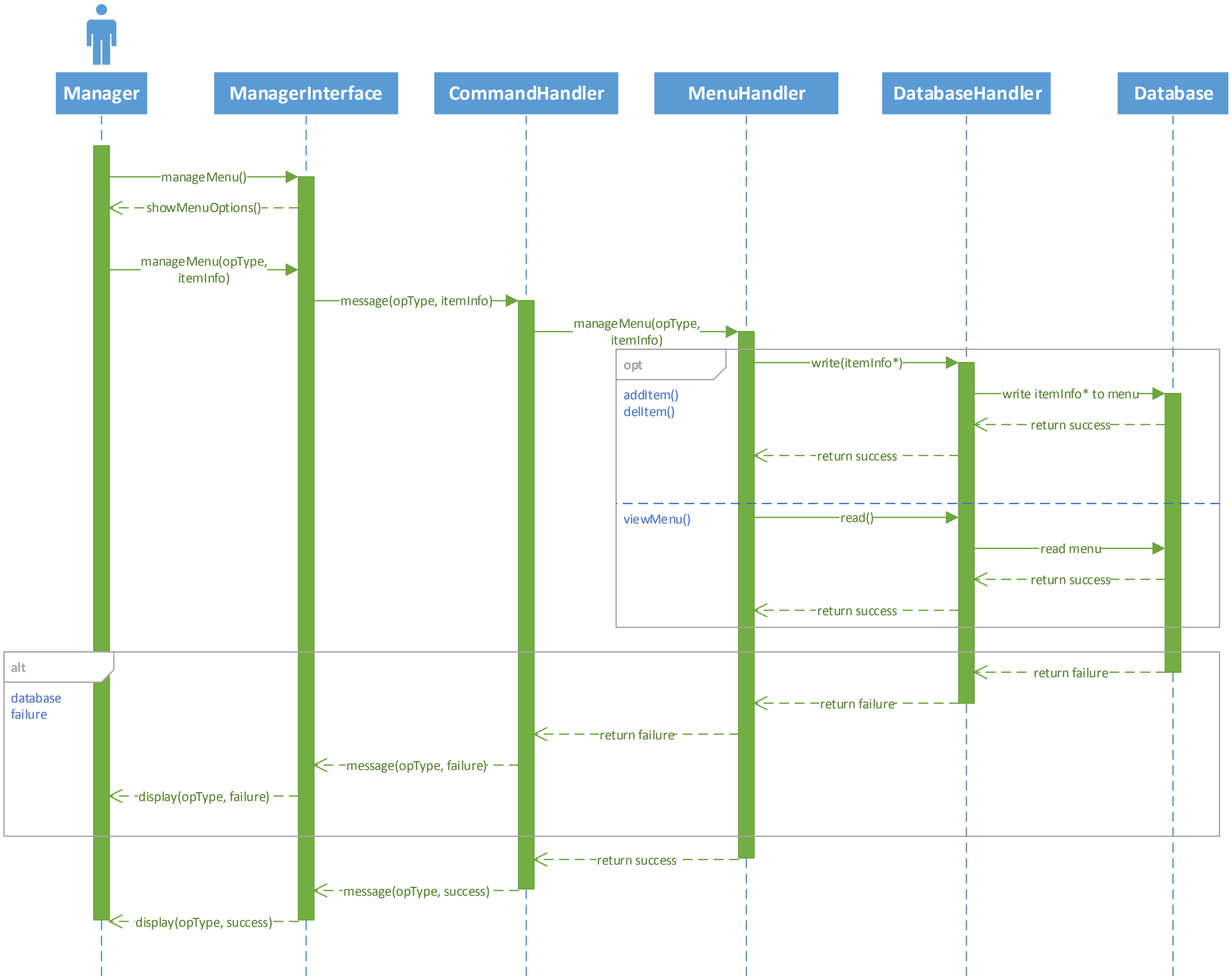
When a menu item is finished cooking, the chef flags that it is done cooking and the scheduling algorithm places it either in the ready queue or the wait queue. It will first search every menu order that is in the wait queue to see if the item corresponds to any of the table orders in the wait queue. If it finds the item in a menu order, then it will increment the number of ready of items that are ready for that table order. If all the items are ready after this one has been added then the table order can be added to the ready queue. if the menu item could not be found in any of the table through the boolean foundTable, then we know what the item either belongs to a singular order or the customer requested his items to come in a first come first serve fashion and therefore can be directly placed onto the ready queue.

# 6. Interaction Diagrams

## 6.1 Interaction Diagrams

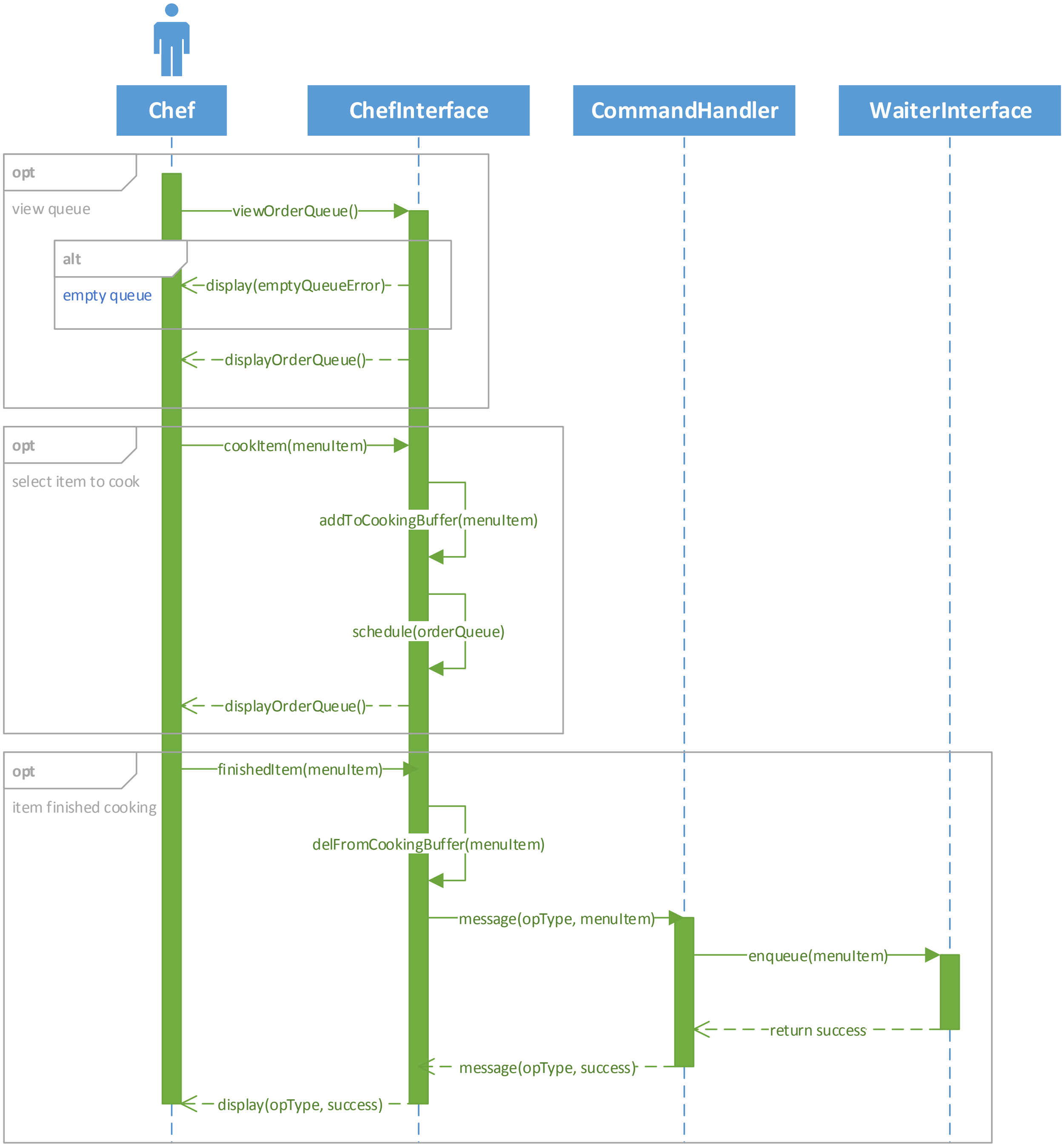


Sequence diagram for UC-1, 2, 3, 4

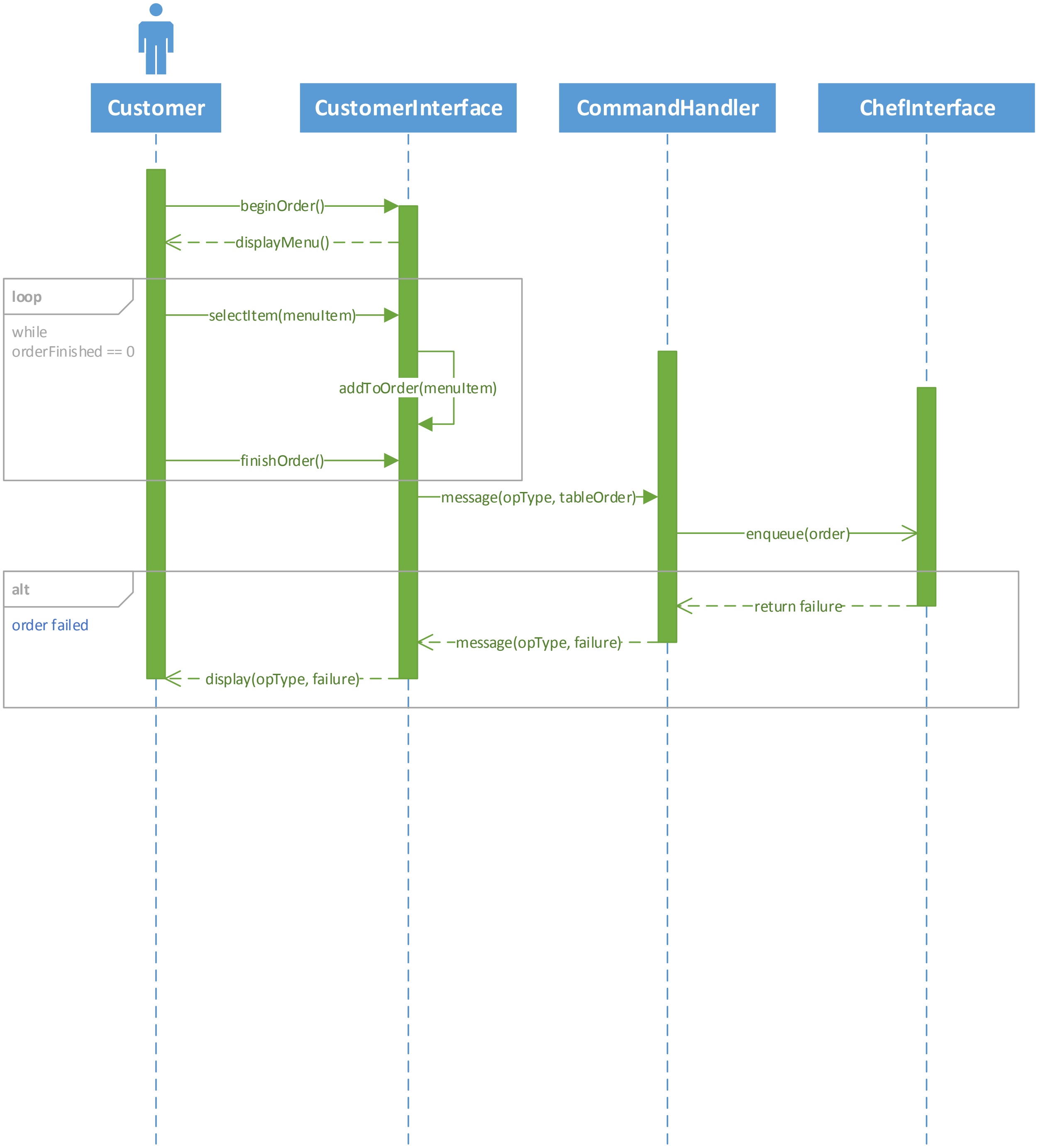


Sequence diagram for UC-8, 9, 10, 11, 12, 13

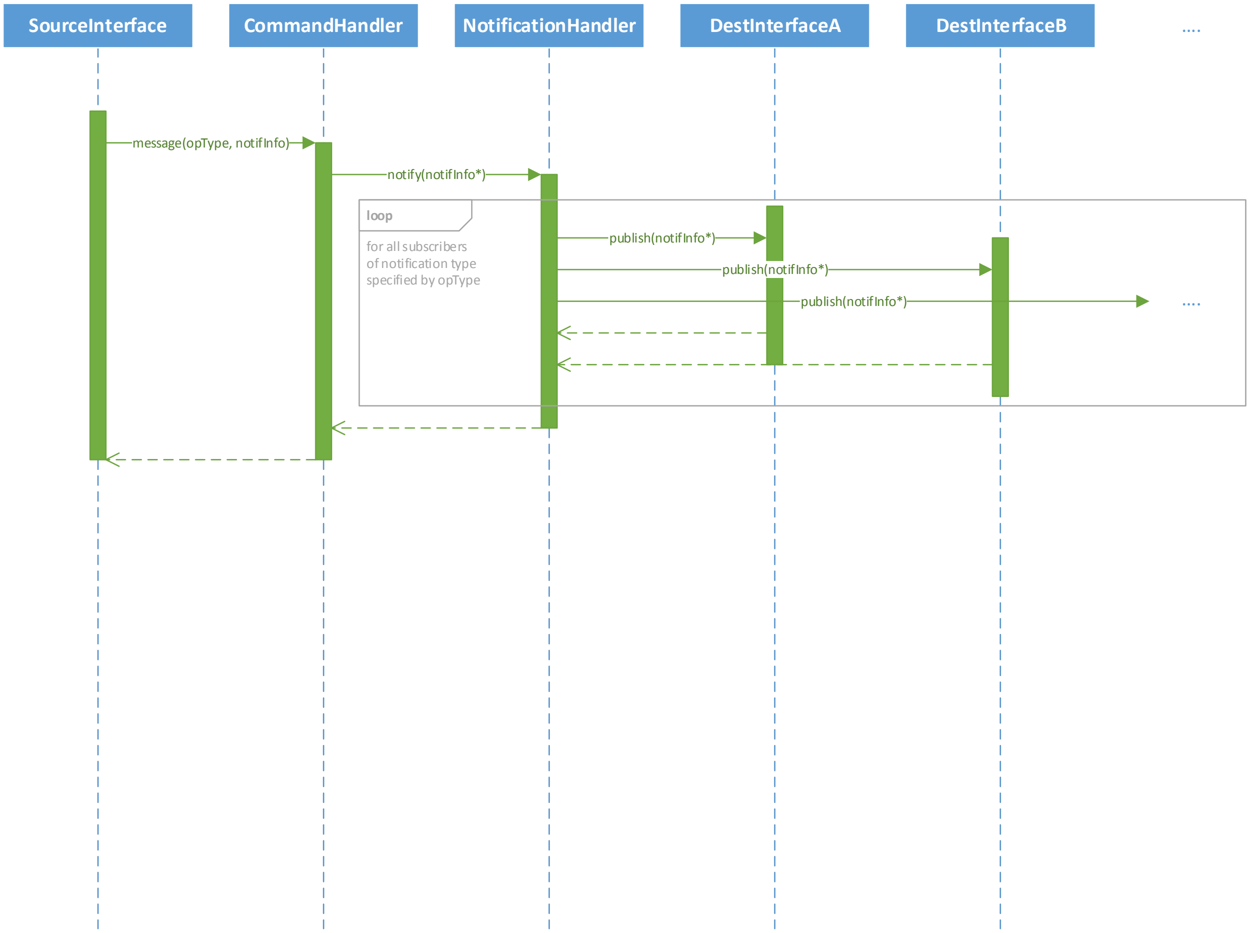
The two sequence diagrams above display the command design pattern. The CommandHandler class is the Command object, the ManagerInterface is the invoker, and InventoryHandler and MenuHandler are respectively the receivers. More information on design patterns can be found in section 7.4.



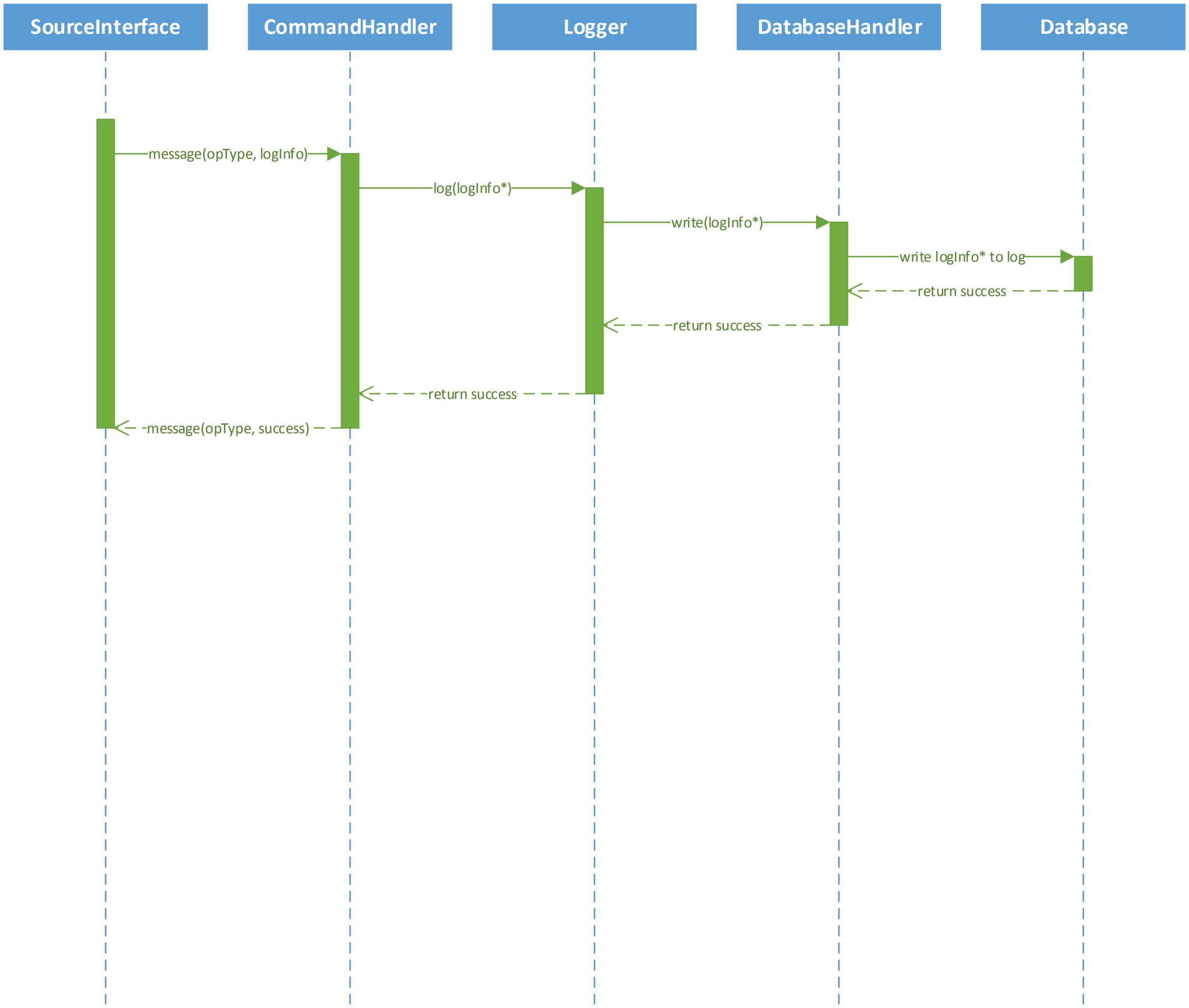
Sequence diagram for UC-14, 15, 16, 17



Sequence diagram for UC-18



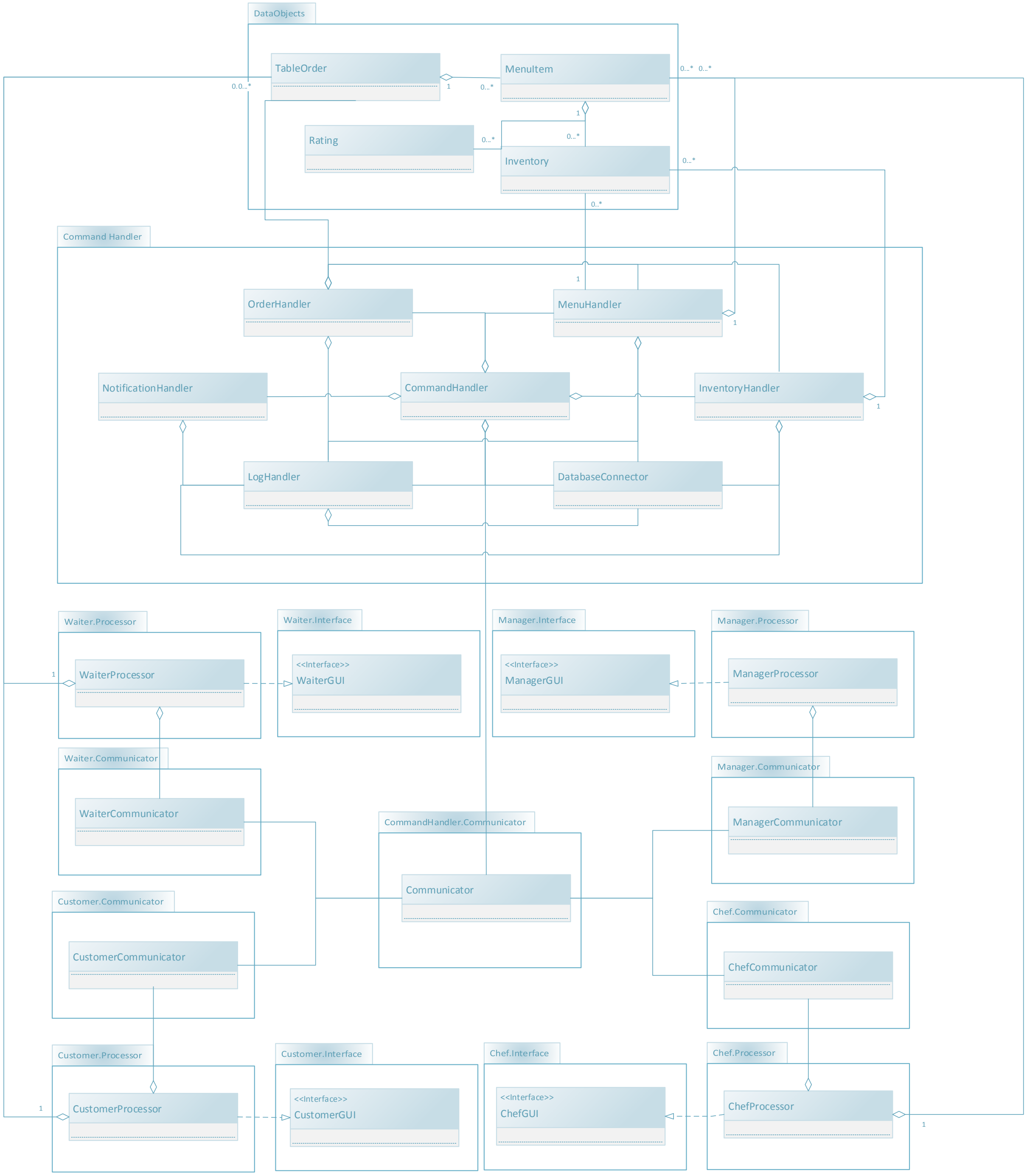
Sequence diagram demonstrating publisher-subscriber notifications



Sequence diagram for logging operations

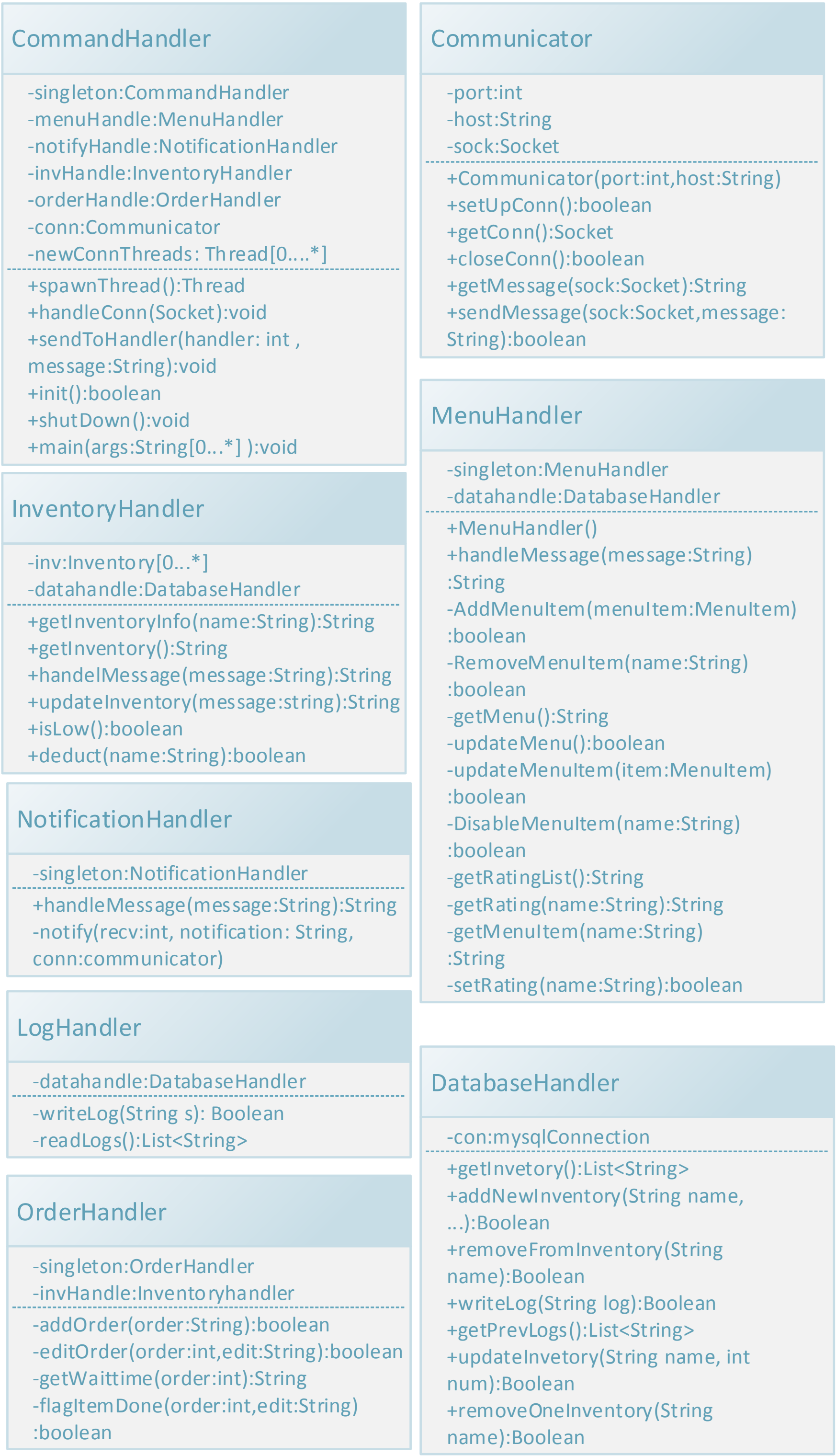
# 7. Class Diagram and Interface Specification

## 7.1 Class Diagram



# 7.2 Data Types and Operation Signatures

## CommandHandler:



```
CommandHandler
-singleton:CommandHandler
-menuHandle:MenuHandler
-notifyHandle:NotificationHandler
-invHandle:InventoryHandler
-orderHandle:OrderHandler
-conn:Communicator
-newConnThreads: Thread[0....*]
+spawnThread():Thread
+handleConn(Socket):void
+sendToHandler(handler: int ,
message:String):void
+init():boolean
+shutDown():void
+main(args:String[0...*] ):void
```

```
InventoryHandler
-inv:Inventory[0...*]
-datahandle:DatabaseHandler
+getInventoryInfo(name:String):String
+getInventory():String
+handelMessage(message:String):String
+updateInventory(message:string):String
+isLow():boolean
+deduct(name:String):boolean
```

```
NotificationHandler
-singleton:NotificationHandler
+handleMessage(message:String):String
-notify(recv:int, notification: String,
conn:communicator)
```

```
LogHandler
-datahandle:DatabaseHandler
-writeLog(String s): Boolean
-readLogs():List<String>
```

```
OrderHandler
-singleton:OrderHandler
-invHandle:Inventoryhandler
-addOrder(order:String):boolean
-editOrder(order:int,edit:String):boolean
-getWaittime(order:int):String
-flagItemDone(order:int,edit:String)
:boolean
```

```
Communicator
-port:int
-host:String
-sock:Socket
+Communicator(port:int,host:String)
+setUpConn():boolean
+getConn():Socket
+closeConn():boolean
+getMessage(sock:Socket):String
+sendMessage(sock:Socket,message:
String):boolean
```

```
MenuHandler
-singleton:MenuHandler
-datahandle:DatabaseHandler
+MenuHandler()
+handleMessage(message:String)
:String
-AddMenuItem(menuItem:MenuItem)
:boolean
-RemoveMenuItem(name:String)
:boolean
-getMenu():String
-updateMenu():boolean
-updateMenuItem(item:MenuItem)
:boolean
-DisableMenuItem(name:String)
:boolean
-getRatingList():String
-getRating(name:String):String
-getMenuItem(name:String)
:String
-setRating(name:String):boolean
```

```
DatabaseHandler
-con:mysqlConnection
+getInvetory():List<String>
+addNewInventory(String name,
...):Boolean
+removeFromInventory(String
name):Boolean
+writeLog(String log):Boolean
+getPrevLogs():List<String>
+updateInvetory(String name, int
num):Boolean
+removeOneInventory(String
name):Boolean
```

## CommandHandler

The CommandHandler is responsible for the handling of all the requests that involve the interaction of multiple interfaces in order to complete a task. These Request will be sent by respective processors.

### Attributes

-singleton:CommandHandler

To show that this class has only one instantiation

-menuHandle:MenuHandler

This object is used to handle all requests that involve the menu.

-notifyHandle:NotificationHandler

This object is used to handle all requests that are notifications.

-orderHandle:InventortyHandler

This object is used to handle all request that involve the Inventory.

-conn:Communicator

This object is used to send and receive requests along with setting up the communication for the server

-newConnThreads:Thread[0…*].

This object hold the threads created for each connection (connections can be abstracted to requests received).

### Methods

+spawnThread():Thread

This method spawn a new thread which is created for each connection.

+handleConn(Socket):void

This method handles the request that it received on a socket.

+sendToHandler(handler:int, message:String):void

This method send the request to the appropriate handler.

+init():Boolean

This method initiates the CommandHandler for accepting request and handling them.

+shutdown():void

This terminates the command handler and all processing requests.

+main(args:String[0….*]):void

This method is used to initialize and pass arguments to the CommandHandler.

## MenuHandler

The MenuHandler is responsible for the handling any changes in the menu that requires the change in all the costumer interfaces. This has to be done in a synchronized way as there are multiple customers who need to have the information to be updated when any menu information is updated.

#### Attributes

-singleton:MenuHandler

To show this class has only one instantiation

-datahandle:DatabaseHandler

This object handles all requests made to the database

#### Methods

+MenuHandler()

This is used by external classes to initialize a menuHandler.

+handleMessage(message:String):String

This class is uses to handles the request give to handle.

-AddMenuItem(menuItem:MenuItem):Boolean

This method is used to add Menu Items to the menu.

-RemoveMenuItem(name:String):String

This method is used to remove any Menu Items from the menu.

-getMenu():String

This method is used to get the whole menu list from the database.

-UpdateMenu():boolean

This method is used to update the local view of the menu from the database.

-DisableMenuItem(name:String):boolean

This method is used to disable a menuItem.

-getRatingList():String

This method is used to get an ordered list of all the highly rated menu Items.

-getRating(name:String):Boolean

This method is used to get the rating of a specific menuItem.

-getMenuItem(name:String):String

This method is used to get the menuItem information.

-setRating( name:String):boolean

This method is used to set the rating for a specific menu item.

### InventoryHandler

The InventoryHandler is responsible for the handling any changes or interfaces with the Inventory. This has to be done in a synchronized way as multiple customers can request Inventory information on menu Items while others can change them.

#### Attributes

-inv:Inventory[0...*]

This object is used to hold all the inventoried list and their descriptions.

-datahandle:DatabaseHandler

This object is used to handle all the requests made to the server.

### Methods

+getInventoryInfo(name:String):String

This method is used to get the inventoried information.

+getInventory():String

This method is used to get the list of inventoried information.

+handleMessage(message:String):String

This method is used to handle any requests that need handling.

+updateInventory(message:String):String

This method is used to update the information on an inventoried item.

+isLow():Boolean

This method is used to check if any Inventoried Items are low;

+deduct(name:String):Boolean

This method is used to deduct the amount for an inventoried item.

## NotificationHandler

The NotificationHandler is responsible for redirecting requests or notification of alerts to and from any interface. Example of the notifications is, Inventory low notification from the Inventory handler to the Manager GUI.

### Attributes

-singleton:NotificationHandler

This shows that the notification handler is limited to one instantiation

### Methods

+handleMessage(message:String):String

This method is used to handle any request that needs to be handled.

-notify(recv:int, notification:String, conn:communicator)

This method is used to notify the

## LogHandler

The LogHandler is responsible for the logging any changes or requests handled by the CommandHandler. As the CommandHandler is the Mediator of requests and processing done by the system, it can log all the actions that takes place on the system.

### Attributes

-dataHandler: DataHandler

This object Is used to interact with the database to store the logs in persistence storage.

#### Methods

+ writeLog(String s) : Boolean

This function will write the log passed into the database, using the database handler.

+ readLogs() : ArrayList<String>

This function is used to return the list of logs that have previously been written to the logger by interacting with the database.

## OrderHandler

The OrderHandler is responsible for the handling any Order requests that need to be sent to the Chef Interface or the Waiter Interface. An Example of these changes can be the sending or a finished order to the WaiterProcessor.

#### Attributes

#### Methods

## DatabaseHandler

The DatabaseHandler is responsible for being a mediator between the Database and any other handlers. This is needed as the requests need to be synchronized to avoid race conditions involved in the writing and reading by the handlers..

#### *Attributes*

-con:mysqlConnection

This is the connector for the mysql database used for persistent storage.

#### *Methods*

+getInventory():List<String>

This method will get the list of inventory items as a string which then must be parsed later on.

+addNewInventory(String name, ...):Boolean

This method will add an inventory item into the inventory.

+removeFromInventory(String name):Boolean

This method will remove an item from the inventory that matches the name that’s passed as an argument.

+writeLog(String log):Boolean

This method will write to the log a message that’s passed in as a string.

+getPrevLogs():List<String>

This method will return a list of previous logs.

+updateInvetory(String name, int num):Boolean

This method will update the inventory of the inventory item with the name that matches the argument with the number that's passed in.

+removeOneInventory(String name):Boolean

This method will remove one from the counter of the inventory item that matches the name that's passed in.

## CommandHandler.communicator:

### Communicator

The Communicator is responsible being the gateway of receiving and sending requests. This class is responsible for creating the connections between the different communicators and to actively receive and send to their communicators on the sockets it created for them.

#### 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

+Communicator(port:int,host:String)

The constructor used to initialize.

+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.

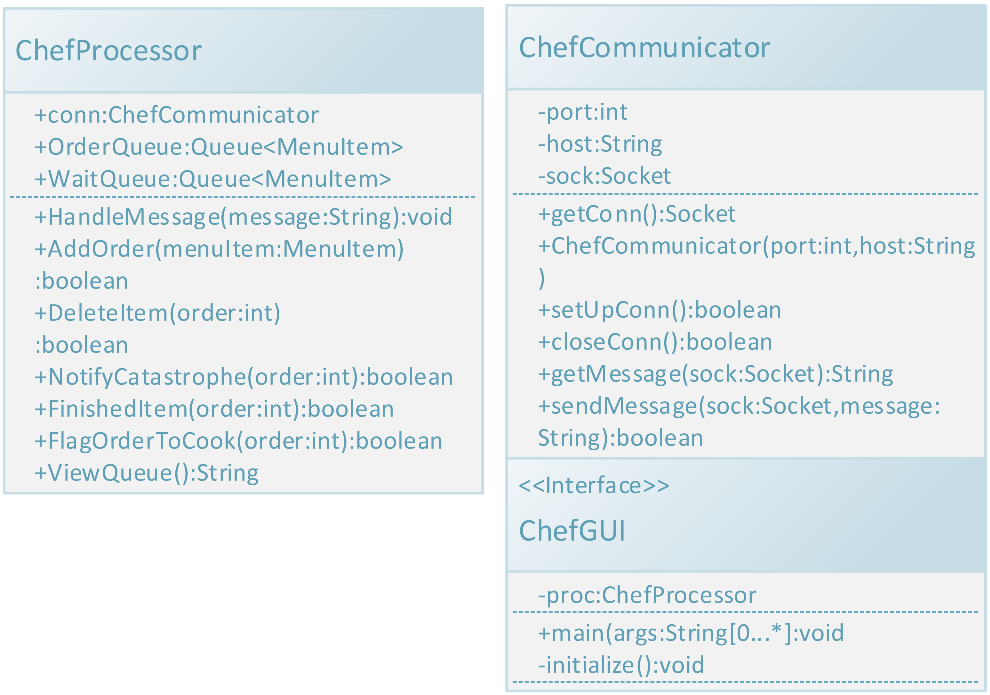
+getMessage(sock:Socket):String

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:


ChefProcessor
+conn:ChefCommunicator
+OrderQueue:Queue<MenuItem>
+WaitQueue:Queue<MenuItem>
+HandleMessage(message:String):void
+AddOrder(menuItem:MenuItem)
:boolean
+DeleteItem(order:int)
:boolean
+NotifyCatastrophe(order:int):boolean
+FinishedItem(order:int):boolean
+FlagOrderToCook(order:int):boolean
+ViewQueue():String
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
<<Interface>>
ChefGUI
-proc:ChefProcessor
+main(args:String[0...*]:void
-initialize():void

## Chef.Processor:

### ChefProcessor

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

#### 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.

+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:

### ChefCommunicator

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.

#### 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.

+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:

### ChefGUI

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

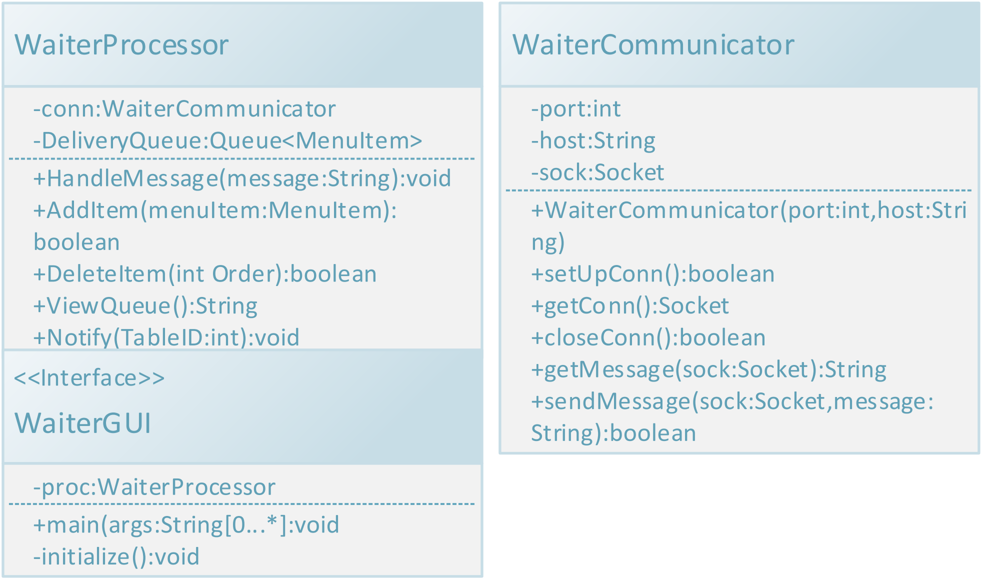
+main(args:String[0….*]):void

This method is used to initialize the GUI.

-intialize():void

This method creates the GUI.

## Waiter



WaiterProcessor

-conn:WaiterCommunicator
-DeliveryQueue:Queue<MenuItem>

+HandleMessage(message:String):void
+AddItem(menuItem:MenuItem): boolean
+DeleteItem(int Order):boolean
+ViewQueue():String
+Notify(TableID:int):void

<<Interface>>
WaiterGUI

-proc:WaiterProcessor

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

WaiterCommunicator

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

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

## Waiter.Processor:

### WaiterProcessor

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

Attributes

-conn:WaiterCommunicator

This is the socket connections used to communicate with other components of the system.

-DeliveryQueue:Queue<MenuItem>

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

Methods

+HandleMessage(message:String):void

This method handles all the messages passed to the waiter.

+DeleteItem(int Order):Boolean

This method removes an order form the deliveryqueue that matches in order number passed.

+ViewQueue():ArrayList<MenuItem>

This methods returns the current delivery queue.

+Notify(tableId:int):void

This methods notifies the customer corresponding the table id passed.

## Waiter.Communicator:

### WaiterCommunicator

The WaiterCommunicator is responsible for only receiving any communication between the CommandHandler and the WaiterGUI. This class is actively listening for requests  from the CommandHandler and can be responsible for any changes made in the WaiterGUI.

#### 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

+WaiterCommunicator(port:int,host:String)

The constructor used to initialize.

+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.

## Waiter.Interface:

### WaiterGUI

The WaiterGUI is the front end of the system and is responsible for the interface 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

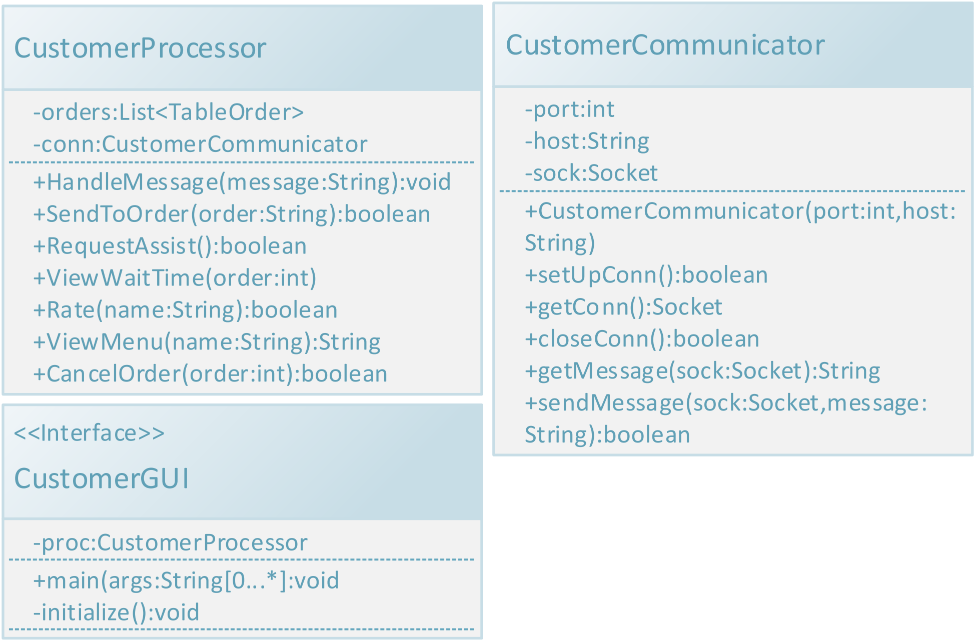
+main(args:String[0....*]):void

This method is used to initialize the GUI.

-intialize():void

This method creates the GUI.

## Customer


CustomerProcessor
-orders:List<TableOrder>
-conn:CustomerCommunicator
+HandleMessage(message:String):void
+SendToOrder(order:String):boolean
+RequestAssist():boolean
+ViewWaitTime(order:int)
+Rate(name:String):boolean
+ViewMenu(name:String):String
+CancelOrder(order:int):boolean
CustomerCommunicator
-port:int
-host:String
-sock:Socket
+CustomerCommunicator(port:int,host:String)
+setUpConn():boolean
+getConn():Socket
+closeConn():boolean
+getMessage(sock:Socket):String
+sendMessage(sock:Socket,message:String):boolean
<<Interface>>
CustomerGUI
-proc:CustomerProcessor
+main(args:String[0...*]:void
-initialize():void

## Customer.Processor:

### CustomerProcessor

The CustomerProcessor is responsible for the maintaining the Order placed by the customer locally while also handing the requests given by the CustomerGUI.

Attributes

-orders:List<TableOrder>

-conn:CustomerCommunicator

This object will be used to communicate with other components of the system using sockets.

Methods

+HandleMessage(message:String):void

This method will handle all message sent to the customer.

+SendToOrder(order:String):Boolean

This method will send an order to the controller to be forwarded to the chef for cooking.

+RequestAssist():Boolean

This method will send a message to the controller to forward a request to the waiter for assistance.

+ViewWaitTime(order:int)

This method will return the wait time for the current order sent.

+Rate(int rating, string comment, name:string):Boolean

This method is used to rate a menu item.

+ViewMenu(name:String):String

This method is used to view the current menu with updated inventory.

+CancelOrder(order:int):Boolean

This method is used to cancel an order with an id matching the one passed into the function.

## Customer.Communicator:

### CustomerCommunicator

The CustomerCommunicator is responsible for only receiving any communication between the CommandHandler and the CustomerGUI. This class is actively listening for requests  from the CommandHandler and can be responsible for any changes made in the CustomerGUI.

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

+CustomerCommunicator(port:int,host:String)

The constructor used to initialize.

+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.

## Customer.Interface:

### CustomerInterface

The CustomerGUI is the front end of the system and is responsible for the interface between the Customer and the system. This class will be using the CustomerProcessor to aid in processing the requests by the Customer.

#### Attributes

-proc:CustomerProcessor

This object is used to process all the requests.

#### Methods

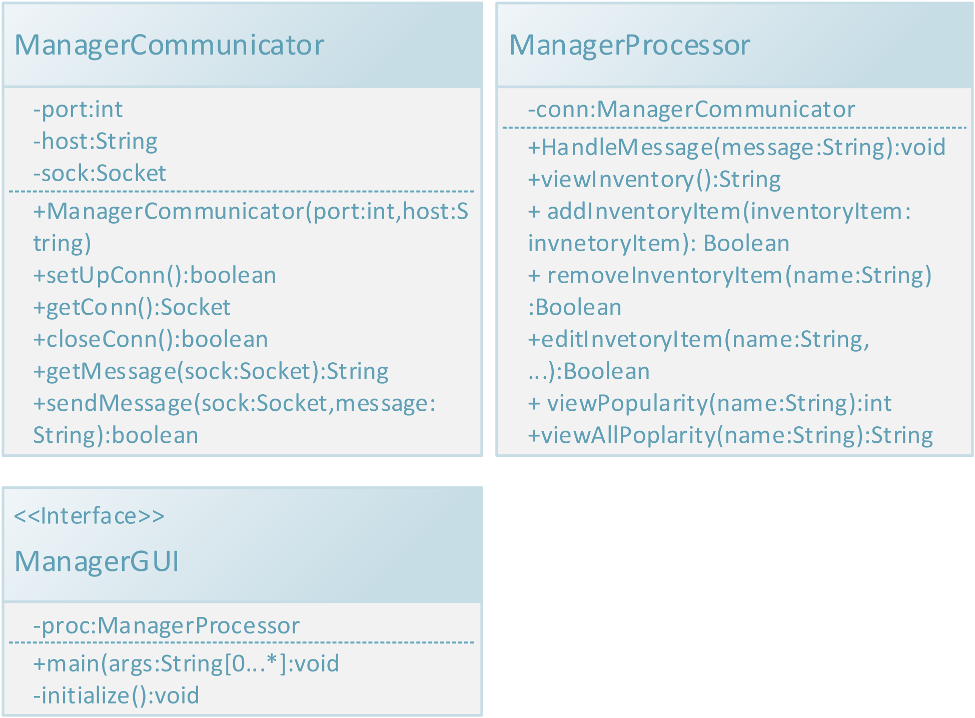
+main(args:String[0....*]):void

This method is used to initialize the GUI.

-intialize():void

This method creates the GUI.

## Manager:


ManagerCommunicator
-port:int
-host:String
-sock:Socket
+ManagerCommunicator(port:int,host:String)
+setUpConn():boolean
+getConn():Socket
+closeConn():boolean
+getMessage(sock:Socket):String
+sendMessage(sock:Socket,message:String):boolean
ManagerProcessor
-conn:ManagerCommunicator
+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
<<Interface>>
ManagerGUI
-proc:ManagerProcessor
+main(args:String[0...*]:void
-initialize():void

## Manager.Processor:

### ManagerProcessor

The ManagerProcessor is responsible for handing the requests given by the CustomerGUI.

Attributes

-conn:ManagerCommunicator

Methods

+HandleMessage(message:String):void

This method handles all the requests it receives.

+viewInventory():String

This method is used to view all the inventoried items of the system.

+addInventoryItem(inventoryItem:Inventory):Boolean

This method is used to add Inventoried Item onto the inventory.

+editInventory(name:String,...):Boolean

This method edits an inventoried Item.

+viewPopularity(name:String):int

This method is used to view popularity of a menuItem.

+viewAllPopularity(name:String):String

This method is used to view an ordered list of all the popular menu Items.

## Manager.Communicator:

### ManagerCommunicator

The ManagerCommunicator is responsible for only receiving any communication between the CommandHandler and the ManagerGUI. This class is actively listening for requests  from the CommandHandler and can be responsible for any changes made in the ManagerGUI.

#### 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

+ManagerCommunicator(port:int,host:String)

The constructor used to initialize.

+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.

## Manager.Interface:

### ManagerGUI

The ManagerGUI is the front end of the system and is responsible for the interface between the Manager and the system. This class will be using the ManagerProcessor to aid in processing the requests by the Manager.

#### Attributes

-proc:ManagerProcessor

This object is used to process all the requests.

Methods

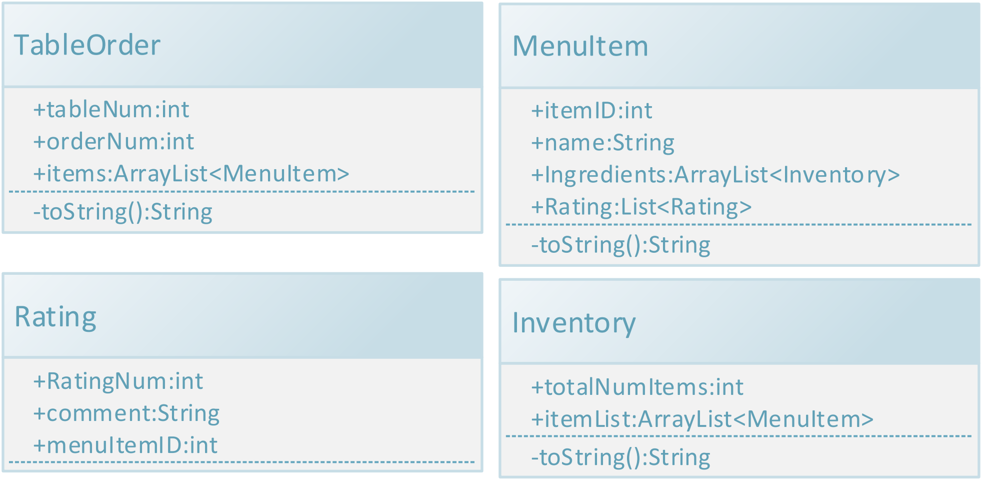
+main(args:String[0….*]):void

This method is used to initialize the GUI.

-intialize():void

This method creates the GUI.

## DataObjects:



TableOrder

+tableNum:int
+orderNum:int
+items:ArrayList<MenuItem>
-toString():String

MenuItem

+itemID:int
+name:String
+Ingredients:ArrayList<Inventory>
+Rating:List<Rating>
-toString():String

Rating

+RatingNum:int
+comment:String
+menuItemID:int

Inventory

+totalNumItems:int
+itemList:ArrayList<MenuItem>
-toString():String

### Inventory

This data object is used to store the Inventory information of an Inventoried item.

Attributes

+totalNumItems:int

This object represents the total number of items in the inventory.

+itemList:ArrayList<String>

This object represents the list of items as strings.

Methods

-toString():String

This method will represent the inventory as a string to be used for displaying the inventory.

### MenuItem

This Data object is used to store the Menu Item information of any dish on the Menu.

#### Attributes

+itemID:int
This object represents the unique id for this menu item.
+name:String
This object represents the name of the menu item being represented.

+Ingredients:ArrayList<String>
This object represents the list of ingredients as strings.

+Rating:list<rating>
This object represents the list of ratings for this menu item.

#### Methods

+toString():String

### TableOrder

This Data object is used to store the TableOrder placed by the customer.

#### Attributes

+tableNum:int
This object represents the table number for this order.

+orderNum:int
This object represents the unique order number for this order.

+items:ArrayList<MenuItem>
This object represents the list of menu items associated with this order.

#### Methods

+toString():String
This method returns a string of the object to easily display the contents.

### Rating

This Data object is used to store the TableOrder placed by the customer.

#### Attributes

+int ratingNum
This object represents the rating from 1-5

+String comment
This object represents the comment for this menu item.

+int menuItemID
This object represents the menu item id associated with this rating.

## 7.3 Traceability Matrix

| | TableOrder | MenuItem | Inventory | OrderHandler | MenuHandler | NotificationHandler | CommandHandler | InventoryHandler | LogHandler | DatabaseConnector | WaiterProcessor | WaiterGUI | WaiterCommunicator | ManagerProcessor | ManagerGUI | ManagerCommunicator | CustomerProcessor | CustomerGUI | CustomerCommunicator | ChefProcessor | ChefGUI | ChefCommunicator | Communicator |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| C1 | X | X | | X | X | | X | | | | | | | | | | X | X | X | | | | X |
| C2 | | X | X | | X | | X | X | | X | | | | | | | | | | | | | X |
| C3 | | | | | | | X | | | | X | X | X | | | | X | X | X | | | | X |
| C4 | X | | | | | | X | | | | | | | | | | | X | X | X | | X | X |
| C5 | | | | | | | | | | | X | X | X | | | | | | | | | | X |
| C6 | | | | | | | X | | | | X | X | X | | | | X | | X | | | | X |
| C7 | | X | | | | | X | | | | | | | | | | | | | X | X | X | X |
| C8 | | | X | | | X | | X | | X | X | X | X | | | | | | | X | X | X | X |
| C9 | | | X | | | | | X | | X | | | | | | | | | | | | | X |
| C10 | | | X | | | X | | X | | X | | | | | | | | | | | | | X |
| C11 | | | X | | | | X | X | | X | | | | X | X | X | | | | X | X | X | X |
| C12 | | X | | | X | | X | | | X | | | | X | X | X | | | | | | | X |

| | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| C13 | | | | | | | X | | X | | | | X | | | X | | | X | | | X | X |

## 7.4 Design Patterns

Our system utilizes design patterns in two major areas. The class commandHandler implements the command design pattern. It receives messages and uses the information encapsulated within those messages to execute methods of other handler or interface classes. We did not implement unexecute functionality because it would be unneccessary for our system -- undoing a placed order is implemented through the editOrder() call, and undoing any other event would make little sense. It is not possible to un-finish a cooked dish, or un-deliver a dish to the customer. We chose this pattern to implement the core communications protocol of our system because the controller and user interfaces will be running on separate machines. The command pattern can be easily implemented using sockets, and centralizing communication around a robust hub reduces the number of potential points of failure compared to a system where each individual machine has a separate line of communication to each other machine. A side benefit is that most error handling can be done on the controller's machine as well. Altogether, this design promotes high cohesion and low coupling for the user interface classes at the cost of a rather large and complex controller module which is further split into its sub-classes.

The notification system uses the publisher-subscriber pattern to deliver notifications to all of the appropriate parties. When a notification is generated and sent to notificationHandler, the handler can parse the type of notification, its content, and its source and determine which interfaces need to receive the notification. It then publishes the notification out to those interfaces. For example, when the system detects that the stock level of an item in the inventory is empty, it will generate a notification which includes a "notification ID" that identifies the type of notification and also any other necessary information; in this case, it would pass the name of the item whose stock has run out. notificationHandler reads the ID and determines that it is a no-stock notification. It reads the input information and constructs the full notification message (e.g. "Warning: Dried pineapple slices are now out of stock"), then publishes the notification to the chefInterface and managerInterface, the two parties who would need to see the notification. The notificationHandler will know each notification type and choose which subscribers should be notified. While notifications could have been implemented using direct communication, using the publisher subscriber pattern makes the system easier to maintain and update. If another user interface must be added for a new employee class, it is simple to include it in the subscribers list and include it in the notification system.

## 7.5 Object Constraint Language

### CommandHandler:

### CommandHandler

1. Invariants: There are messages which need to be processed.
   context CommandHandler
   inv: self.socket != NULL
2. Pre-conditions: There is at least one message which is sent to the CommandHandler.
   context CommandHandler : int
   pre: self.numberofthreads > 0
3. Post-conditions: The message(s) have been processed.
   context CommandHandler : int
   post: self.numberofthreads = 0

### MenuHandler

1. Invariants: An action affecting the menu has been invoked.
   context MenuHandler
   inv: self.command != NULL
2. Pre-conditions: There is at least one action involving the menu that must be handled.
   context MenuHandler : int
   pre: self.numberofactions > 0
3. Post-conditions: The action(s) have been processed
   context MenuHandler : int
   post: self.numberofactions = 0

### InventoryHandler

1. Invariants: An action affecting the inventory system has been invoked.
   context InventoryHandler
   inv: self.command != NULL
2. Pre-conditions: There is at least one action involving the inventory that must be handled.
   context InventoryHandler : int

pre: self.numberofactions > 0

3. Post-conditions: The action(s) have been processed
   context InventoryHandler : int
   post: self.numberofactions = 0

## NotificationHandler

1. Invariants: The notification handler receives a notification.
   context NotificationHandler
   inv: self.notification != NULL
2. Pre-conditions: There is at least one notification that must be handled.
   context NotificationHandler : int
   pre: self. numberofnotifications > 0
3. Post-conditions: The action(s) have been processed
   context NotificationHandler : int
   post: self.numberofnotifications = 0

## LogHandler

1. Invariants: The log handler receives data to write into the log
   context LogHandler
   inv: self.data != NULL
2. Pre-conditions: There is at least one thing to be written into the log
   context LogHandler : int
   pre: self.numberoflogentries > 0
3. Post-conditions: The entries have been written into the log
   context LogHandler
   post: self.numberoflogentries = 0

## OrderHandler

1. Invariants: The log handler receives data to write into the log.
   context LogHandler
   inv: self.data != NULL
2. Pre-conditions: There is at least one thing to be written into the log.
   context LogHandler : int
   pre: self.numberoflogentries > 0
3. Post-conditions: The entries have been written into the log.
   context LogHandler : int
   post: self.numberoflogentries = 0

## DatabaseHandler

1. Invariants: There exists a pending action involving the database.
   context DatabaseHandler
   inv: self.command != NULL
2. Pre-conditions: The database has not yet been written to.
   context DatabaseHandler : Boolean
   pre: self.databasewritten == false
3. Post-conditions: The database has been updated/modified.
   context DatabaseHandler : Boolean
   post: self. databasewritten == true

## CommandHandler.communicator:

### Communicator

1. Invariants: There is a socket connected to a client.
   context Communicator
   inv: self.socket != NULL
2. Pre-conditions: There is a message that needs to be sent or received.
   context Communicator : int
   pre: self.messagequeue.size() == 1
3. Post-conditions: The message is sent.
   context Communicator : int
   post: self.messagequeue.size() == 0

### ChefProcessor

1. Invariants: An action is performed affecting the items in WaitQueue.
   context ChefProcessor
   inv: self.socket != NULL
2. Pre-conditions: The customer submits and order or the chef removes an order.
   context ChefProcessor : Boolean
   pre: self.actionexists == true
3. Post-conditions: WaitQueue is updated.
   context ChefProcessor : Boolean
   post: self.actionperformed == true

### ChefCommunicator

1. Invariants: An action has been performed on the WaitQueue and needs to be communicated to the central controller.
   context ChefCommunicator
   inv: self.socket != NULL

2. Pre-conditions: A message from the chef GUI needs to be sent.
   context ChefCommunicator : Boolean
   pre: self.messageexists == true
3. Post-conditions: The message is sent.
   context ChefCommunicator : Boolean
   post: self.messagesent == true

## ChefGUI

1. Invariants: WaitQueue has been updated.
   context ChefGUI
   inv: self.socket != NULL
2. Pre-conditions: WaitQueue has been modified and display needs to be refreshed.
   context ChefGUI : Boolean
   pre: self.waitqueueupdate == true
3. Post-conditions: The interface is displayed.
   context ChefGUI : Boolean
   post: self.displaysuccess == true

## Waiter

## WaiterProcessor

1. Invariants: An action is performed affecting the items in the DeliveryQueue.
   context WaiterProcessor
   inv: self.socket != NULL
2. Pre-conditions: Chef finishes cooking an order and marks it ready for delivery or the Waiter delivers an item and removes it from the queue.
   context WaiterProcessor : Boolean
   pre: self.actionexists == true
3. Post-conditions: The DeliveryQueue is updated.
   context WaiterProcessor : Boolean
   post: self.actionperformed == true

## WaiterCommunicator

1. Invariants: An action has been performed on the DeliveryQueue and needs to be communicated to the central controller.
   context WaiterCommunicator
   inv: self.socket != NULL
2. Pre-conditions: A message from the waiter GUI needs to be sent.
   context WaiterCommunicator : Boolean

pre: self.messageexists == true

3. Post-conditions: The message is sent.
   context WaiterCommunicator : Boolean
   post: self.messagesent == true

## WaiterGUI

1. Invariants: WaitQueue has been updated.
   context WaiterGUI
   inv: self.socket != NULL
2. Pre-conditions: DeliveryQueue has been modified and display needs to be refreshed.
   context WaiterGUI : Boolean
   pre: self.deliveryqueueupdate == true
3. Post-conditions: The interface is displayed.
   context WaiterGUI : Boolean
   post: self.displaysuccess == true

## Customer

## CustomerProcessor

1. Invariants: An action is performed affecting the items in the OrderQueue.
   context CustomerProcessor
   inv: self.socket != NULL
2. Pre-conditions:  Customer submits an order.
   context CustomerProcessor : Boolean
   pre: self.actionexists == true
3. Post-conditions: The OrderQueue is updated.
   context CustomerProcessor : Boolean
   post: self.actionperformed == true

## CustomerCommunicator

1. Invariants: An action has been performed on the OrderQueue and needs to be communicated to the central controller.
   context WaiterCommunicator
   inv: self.socket != NULL
2. Pre-conditions: A message from the customer GUI needs to be sent.
   context WaiterCommunicator : Boolean
   pre: self.messageexists == true
3. Post-conditions: The message is sent.
   context WaiterCommunicator : Boolean

post: self.messagesent == true

### CustomerGUI

1. Invariants: OrderQueue has been updated.  
   context CustomerGUI  
   inv: self.socket != NULL
2. Pre-conditions: OrderQueue has been modified and display needs to be refreshed.  
   context CustomerGUI : Boolean  
   pre: self.orderqueueupdate == true
3. Post-conditions: The interface is displayed.  
   context CustomerGUI : Boolean  
   post: self.displaysuccess == true

### Manager

### ManagerProcessor

1. Invariants: An action is performed affecting the items in the inventory database.  
   context ManagerProcessor  
   inv: self.socket != NULL
2. Pre-conditions: Manager performs a restock action on the inventory system or Customer places an order, which should decrease the ingredient count in the inventory.  
   context ManagerProcessor : Boolean  
   pre: self.actionexists == true
3. Post-conditions: The inventory database is updated.  
   context ManagerProcessor : Boolean  
   post: self.actionperformed == true

### ManagerCommunicator

1. Invariants: An action has been performed on the inventory database and needs to be communicated to the central controller.  
   context ManagerCommunicator  
   inv: self.socket != NULL
2. Pre-conditions: A message from the manager GUI needs to be sent.  
   context ManagerCommunicator : Boolean  
   pre: self.messageexists == true
3. Post-conditions: The message is sent.  
   context ManagerCommunicator : Boolean  
   post: self.messagesent == true

### ManagerGUI

1. Invariants: The inventory database has been updated.
   context ManagerGUI
   inv: self.socket != NULL
2. Pre-conditions: The inventory database has been modified and display needs to be refreshed.
   context ManagerGUI: Boolean
   pre: self.orderqueueupdate == true
3. Post-conditions: The interface is displayed.
   context ManagerGUI: Boolean
   post: self.displaysuccess == true

### Data Objects:

### Inventory

1. Invariants: N/A
2. Pre-conditions: N/A
3. Post-conditions: N/A

### MenuItem

1. Invariants: N/A
2. Pre-conditions: N/A
3. Post-conditions: N/A

### TableOrder

1. Invariants: N/A
2. Pre-conditions: N/A
3. Post-conditions: N/A

# 8. System Architecture & System Design

## 8.1 Architectural Styles

Due to complex architectural nature of our project, we chose to use several patterns together to model the system. We can categorize the structure our system by identifying the main parts, and then apply architectural considerations with respect to each piece.

Our system is as follows:

- People (Manager, Chef, Waiter, Customer)
- Controller
- Database

### 8.1.1: Client/Server Architecture

Our system requires persistent data storage to maintain the records and inventory for the restaurant, and constant access to this stored data. Therefore, we require a server, and with that a **client/server** architecture. On the server side, we have the Database subsystem which contains the persistent information for our restaurant. This server can handle the requests from the client and send or receive information as needed. The client side contains the Controller subsystem, which has access to the information in the Database. In our system, the Controller and the Database reside on the same computer, and is implemented with MySQL. MySQL allows us to create a local server that takes a chunk of the hard drive, and through communication protocols we can use any common language (such as Java) to access the stored information. Utilizing this software, we can eliminate the need to a dedicated server computer and complicated communication protocols, making our product cheaper and easier to implement.

### 8.1.2: Event-Driven Architecture

Event-driven architecture (EDA) is a software pattern promoting the production, detection, consumption of, and reaction to events [3]. An event is defined as a significant change in the state of an object in the system. This pattern consists typically of event emitters and consumers. Event emitters are parts of the system which could trigger events, and consumers are the parts of the system that react to these events.

In our system, we are utilizing our notification messages as states. When an emitter sends a message, the controller (which acts as the event processing engine), handles the message and evokes the appropriate response from the event consumer. The customer and inventory are event generators, while the waiter, chef, and manager are consumers. For example, when the help button is pressed at the customer's computer, it sends a message to the controller which will then interpret it as a help message and pass it off to the waiter, who will provide assistance. Another example is when an item in the inventory falls below the amount threshold, the inventory handler will send a notification to the controller, which will interpret the message as a low-ingredient warning and pass it off to the manager.

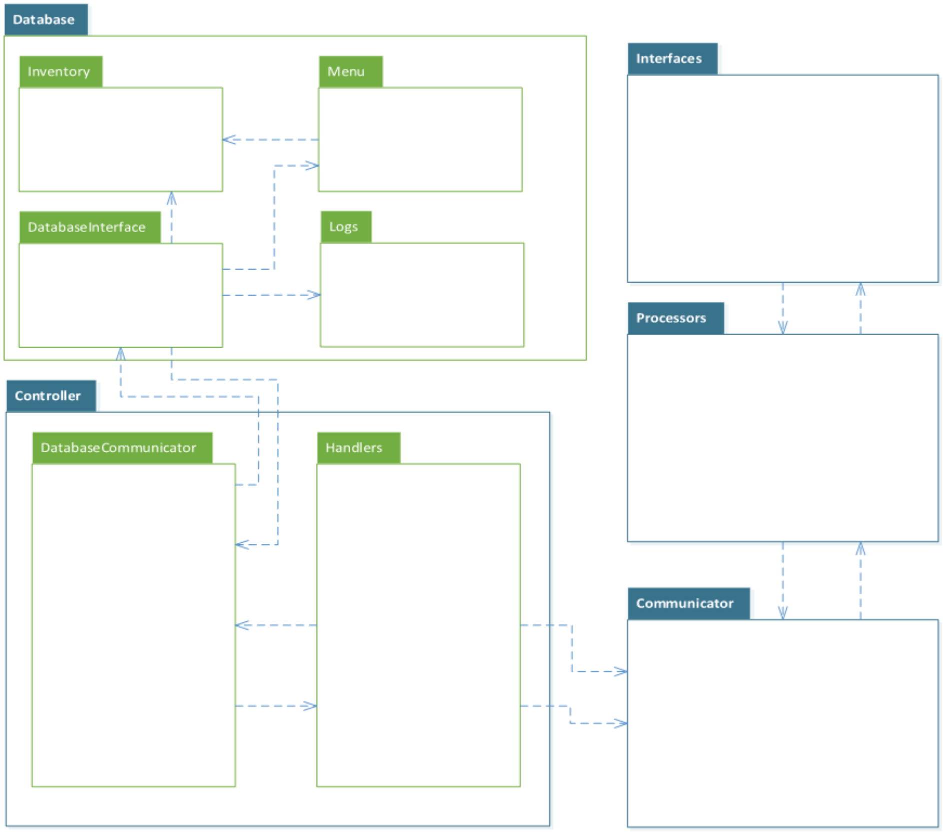
### 8.1.3: Object-Oriented Architecture

Object-oriented architecture is a design paradigm based on the division of responsibilities for an application or system into individual reusable and self-sufficient objects, each containing the data and the behavior relevant to the object [4]. Within an objected-orientated system, objects cooperate amongst each other to complete tasks and form the system. These objects are independent and loosely-coupled modules which communicate by sharing certain datum or methods, in addition to sending and receiving messages.

Our system is heavily object-oriented. At the core, all of our objects are separate modules- the Chef Interface, Waiter Interface, Manager Interface, and Customer Interface objects are all derived from the a common class, since they all share basic methods of displaying information and communicating with the controller. The Controller object is composed of several other classes which regulate the system and function independently but work to compose the whole. Our menu is composed of individual menu item objects, which contain their own attributes.

From the bottom up, our system was designed with object-oriented programming in mind, and our design choices reflect this.

## 8.2 Identifying Subsystems



This UML Package Diagram compartmentalizes the general responsibilities of our subsystems into singular objects in order to more easily describe our system. The controller archetype contains a major object, called the Handler, which regulates most of the tasks in our system. It has a co-dependency with DatabaseCommunicator, which talks with DatabaseInterface. This interface is an access point for the information stored in the database including Logs, Inventory, and Menu (which is dependent on the inventory). The handler also has a co-dependency with the Communicators object, which delegates information to Processors. The processors communicate directly with Interfaces, which control the GUIs of the system. The processors control the underlying business logic, while the interfaces display it. The Interfaces and Processors objects are supertypes of processors/interfaces needed for the Manager, Chef, Waiter, and Customer.

## 8.3 Mapping Subsystems to Hardware

The Subsystem diagram, as seen in the above figure, can easily be mapped to our hardware specifications.

- The Database and Controller are allocated on the central computer.
    - The Database is a MySQL server running on this central computer.

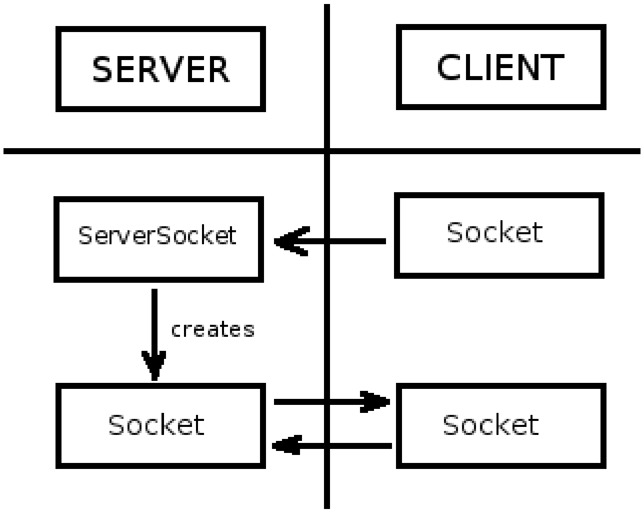
- Each specific actor (Manager, Chef, Waiter, Customer) has a separate tablet computer, which allocate their respective Interfaces, Processors, and Communicators.

## 8.4 Persistent Data Storage

Auto-Serve needs to store data that will outlast a single execution of the system in order to keep track of the different transactions that are going on. The transactions can involve customer orders, inventory restocks, popularity ratings, and any changes made to the menu o the order. For each table in the restaurant, the controller will log all the transaction into the database. The database will be maintained and implemented using MySQL. The controller will be communicating with the database only and all other terminals will retrieve data from the database through the controller as the middle person. Updates to the existing tables in the database occur when a new transaction is made or an old transaction is edited. Storing the data in a relational database will allow for efficient querying and manipulation of data to needs of each particular client module (terminal).

## 8.5 Network Protocol

Auto-Serve will utilize Java Sockets built upon TCP/IP protocol. The terminals within the restaurants will be communicating with the controller using this protocol. The controller will act as a server and every other module (terminal) as a client. The algorithms that are being used in the restaurant communicate between different modules (terminals) in the system by Java sockets that are created on top of the TCP/IP protocol. The messages we will be sending through the sockets will be object messages. We made the choice to use this due us programming our demo in Java.  A diagram of how this should look like is below:



## 8.6 Global Control Flow

### 8.6.1 Execution Orderness

Auto-Serve is procedure-driven and everything executes in a linear fashion. The linear fashion can be described as followed:

Each customer that comes in will follow the same pattern of ordering menu items from their respective table. Afterwards, the chef will be following the same pattern of choosing menu items from the order queue and flagging them when the food is done cooking. The waiter then will be notified of the food item that is done cooking, so he can deliver it. The manager will be doing a daily routine check on the system ensuring all is properly functioning within the restaurant.

### 8.6.2 Time Dependency

Auto-Serve is event-response time for the automation of our inventory alerts, but for the rest of the system it is a real-time system. The real-time system is periodic. The scenario that is explained in the Execution Orderness is what is periodic with the customer ordering a menu item, to the chef preparing it, and the waiter delivering it to the customer. All of this is time dependent as the time that the customer orders will be taken into consideration in our algorithm for queuing menu items. Another factor where real-time comes into consideration is wait time where the customer will be able to see how long he will have to wait before he will received his food. The wait time will help the customer decide what food to order and the chef to increase output of the food to the waiter to be delivered to the customer.

The inventory freshness factor is also time dependent. The ingredients inside the inventory will eventually go bad after the expiration time on it. When an inventory item is added into the inventory system the time of expiration is added also to ensure that the ingredient is used up in the preparation of the menu item before the expiration time.

### 8.6.3 Concurrency

Auto-Serve will contain multiple threads, which involves multiple subsystems running independently of each other. All interactions between the subsystems are controlled through the controller. Multiple customers will be placing order at the same time which is one way we need concurrency. This situation of multiple customer can be taken care of by running the different threads through the order queue. Another scenario could be the manager checking the restaurant inventory amount and updating it. This would have to be done by spawning another thread in the controller which would handle this update request in the database. The synchronization of the threads aren't really needed as they would be working independently of each other.

## 8.7 Hardware Requirement

The hardware requirements that are needed for auto-serve are quite simple. There will be four types of different terminals. The terminals will be respectively for the Chef, Customer, Waiter, and Manager. There will be more than one terminal for the Customer as there will be one placed on every table in the restaurant. The controller will be one central server which will be connected to all the terminals through an internal Local Area Network which will send the proper data to the proper terminal as well as log all the transactions that are going on at the same time. The terminals can be in the form of a computer for the manager to view everything that is going on in the restaurant. For the chef it can be an android based tablet attached to the wall where he can either use a keyboard or touch screen to navigate through the different options. For the waiter it can be a touchscreen android tablet embedded in the corner of the restaurant. For the customers it will be touch screen android tablet (one for each table in the restaurant). The tablets for the customers should have a minimum display of 1ft by 6 inches.  An internet connection is needed for Local Area Network communication within the restaurant and for external communication with the inventory vendor and others. The device internal specifications are elaborated below.

### 8.7.1 Controller

The controller should have the following specifications to function properly:

| HARDWARE | MINIMUM REQUIREMENTS |
|---|---|
| PROCESSOR | Intel Xeon E7 |
| RAM | 4 GB |
| HARD DRIVE | 250 GB |
| NETWORK CARD | 10/100/1000Mbps |

### 8.7.2 Computer

The computer should have the following specifications to accomplish daily tasks.

| HARDWARE | MINIMUM REQUIREMENTS |
|---|---|
| PROCESSOR | Intel Core 2 Duo |
| RAM | 4 GB |
| HARD DRIVE | 250 GB |
| NETWORK CARD | 10/100Mbps |
| DISPLAY | 1920 x 1080 Resolution |

### 8.7.3 Tablets

The tablets will vary for the chef, waiter, and the customer. The tablet for the chef and waiter will have the same specifications, but the customer will have different ones.

**Tablets for Waiter/Chef**

| HARDWARE | MINIMUM REQUIREMENTS |
|---|---|
| PROCESSOR | Quad Core |
| MEMORY | 4 GB |
| HARD DRIVE | 32 GB |
| NETWORK CARD | 10/100Mbps or Wifi Enabled |
| DISPLAY | 1024 x 760 Resolution |

**Tablets for Customer**

| HARDWARE | MINIMUM REQUIREMENTS |
|---|---|
| PROCESSOR | Dual Core |
| MEMORY | 2 GB |
| HARD DRIVE | 16 GB |
| NETWORK CARD | 10/100Mbps or Wifi Enabled |
| DISPLAY | 1024 x 760 Resolution |

The resolution has to be above the minimum requirement to ensure that the chef, waiter, and customer are able to read the data presented to them on the tablets properly.

# 9.1 Algorithms

The most noteworthy functions have been already described in the mathematical model section of the report. The algorithms described here are only ones that are noteworthy and should be mentioned for completeness and understanding of how the system works. Some algorithms are either very trivial or a duplicate of one already mentioned and will not be described here.

## 9.1.1 Controller

### 9.1.1.1 Command Handler

The controller is essentially the center of remote procedure call communication. Its subcomponent, the command handler implements a simple algorithm to pass messages to internal components of the controller. Message objects are created for the types of messages in the system. These objects may contain data or may just be made to signal that an event occurred. These messages, shared by all parties in the system, when passed to the command handler, will go through a switch case to determine what component of the controller that it goes to next.

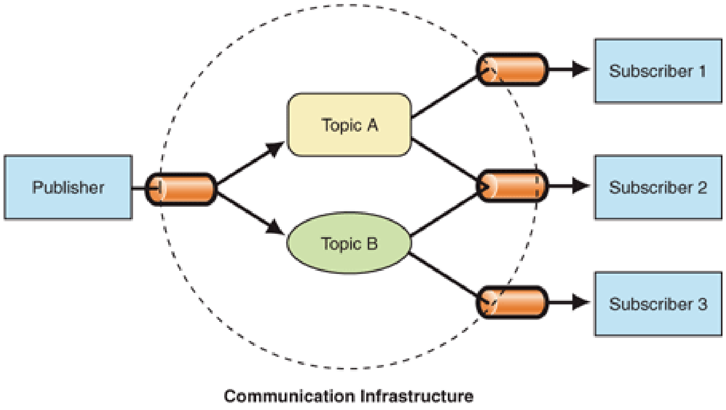
The algorithm is implemented using a switch case.

For example, a message for an order will first go to the command handler from the customer and the command handler, determines its message for an order and then sends this message to the order handler, which will then process it.

```
//for just the order message
Switch(message m) {

Case Message.order:
        orderHandler.process(m);
}
```

### 9.1.1.2 Notification System

The notification system uses a publisher – subscriber design pattern where it is the publisher and the other components subscribe to certain messages that it sends out. The algorithm behind this is the same as any publisher-subscriber system. It can be summarized as



```
For each Message M to be published
    For each Subscriber S of M.type
        Send (notification, S)
```

### 9.1.1.3 Logger

The logger logs events that come through the command handler to the database or a file. There is no specific algorithms that are worth mentioning.

### 9.1.1.4 Database Connector

The database provides an interface to the MYSQL database on the controller's system. More details about MYSQL and how the database is implemented behind the programmer's interface can be found through its creator, Oracle, Inc. This includes algorithms for sorting and manipulating data.

### 9.1.1.5 Inventory Handler

The Inventory handler deals with updating the inventory on successful placements of orders, it does not have any specific algorithms that are worth mentioning since it will only add or subtract a count based on the number of items ordered, number of ingredients used, etc.

### 9.1.1.6 Menu Handler

The menu handlers handles updating the menu list that the customers can order from.  Just as the inventory handler, it will add or remove an item

### 9.1.1.7 Order Handler

The order handler will handle orders that customers create. There are no specific algorithms worth mentioning.

## 9.1.2 Chef

### 9.1.2.1 Java Swing Algorithms for GUI

These algorithms are described in the customer section.

### 9.1.2.2 Queuing orders to be cooked

This algorithm is also described in the mathematical section, but is in more detail here, since it will incorporate the actual elements from other components of the system.

## 9.1.3 Waiter

### 9.1.3.1 Java Swing Algorithms for GUI

These algorithms are described in the customer section.

### 9.1.3.2 Queuing of table orders for waiters

This algorithm is also stated in the theoretical model section.  Before we describe the algorithm for queuing, we need to describe how the wait list gets built.

### 9.1.4 Customer

#### 9.1.4.1 Java Swing Algorithms for GUI

The algorithms used in the customer GUI are quite simple. The GUI itself has predefined buttons, labels, and lists that will dynamically update themselves based on the data that the customer is dealing with. The algorithms behind this is JAVA Swing. Swing has a set of APIs that allow the programmer to interface with the graphical elements. Since the programmer is abstracted from any algorithms used internally in swing, further information can be found through the JAVA website.

## 9.2 Data Structures

Other than the database, which was used for consistent storage during system down time, the main data structures that were used throughout the system were array lists and queues. The queues form the backbone of the chef and waiter components. And Array lists were used through the project for storing data as a list. We chose to use array lists because of its simplicity and because the actual data that it represented performs the best in a list because the operations that are done are always in a first to last order. For instance, the GUIs will always take the list as is and display it on the GUI elements.

In terms of performance of retrieval and traversal, both array and linked lists structures have O(1) insertion and O(n) retrieval. The linked list has an O(1) deletion while the array list has an O(n) deletion. However, we chose to use array lists over linked lists because of their ability to dynamically resize and ability to fit the data better. Because deletion is rarely needed when we store our data, array lists are the better choice.

### 9.2.1 Controller

For the controller, there are no advanced data structures that specifically need to be mentioned. There are temporary structures for instance when dealing with the database, to hold items that are coming to and from the database. A list would be used if the database returns a list. These data structures have no use other than to temporary store data to be passed along to other components. And therefore have no criteria in deciding between performance, flexibility, etc. These structures were the bare minimum to contain the corresponding data.

### 9.2.2 Chef

The chef's primary data structure is his queue. This queue is list of composite objects, namely table orders that are coming from customer. This queue is an array that is dynamically updated every time the scheduling algorithm is applied.

### 9.2.3 Waiter

The waiter's primary data structure is his queue. This queue is essentially the same as chef queue except it contains a different composite object. In this case, it contains table orders as a whole.

### 9.2.4 Customer

The customer's menu is stored in an array of composite menu item objects. Each of these objects will contain information that the customer GUI will display about the item. This list is also an array.

# 10. User Interface Design and Implementation

The user interfaces have not changed much from our initial mock up developed in report 1. All the functionality that we have discussed in the report are illustrated in the mock up that were developed for report 1.  The use cases that we will be implementing for demo 2 are the following in which we will be bringing the integration of the different GUI's together for demo 2.

Use Cases to be implemented for demo 2:

UC – 1 ManageInventory
UC – 2 ViewInventoryList

UC – 3 AddInventoryItem (did not implement UC – 4 RemoveInventoryItem)
UC – 8 ManageMenu
UC – 9 AddMenuItem (did not implement UC – 10 RemoveMenuItem)
UC – 13 ViewMenu
UC – 14 ManageOrders
UC – 15 ViewOrderQueue
UC – 16 SelectOrderToCook
UC – 17 FlagOrderDone
UC – 18 PlaceOrder (we implemented two versions: Place Order FIFO, Place Order As Group)
UC – 20 Log
UC – 22 RequestWaiter
UC – 26 SendNotification

The new GUI's that we demonstrated for demo 2 are shown below.

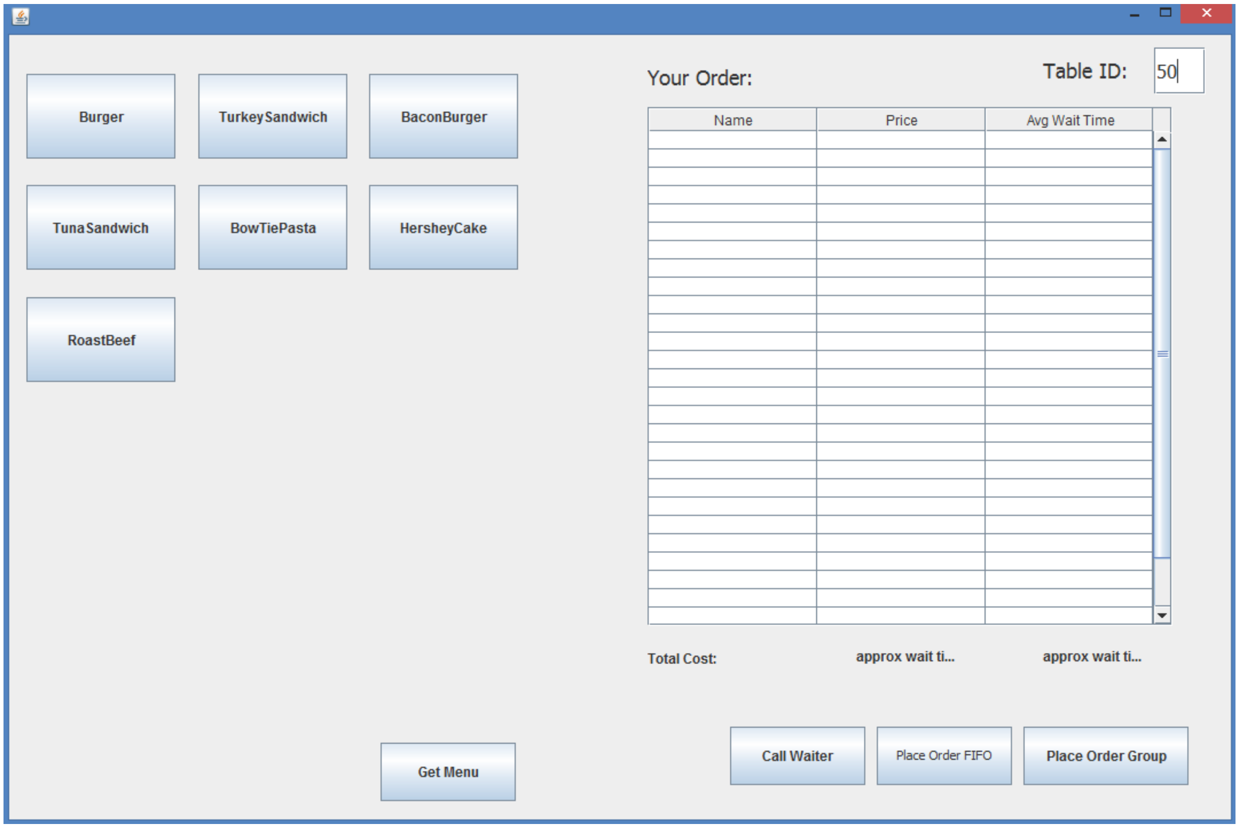
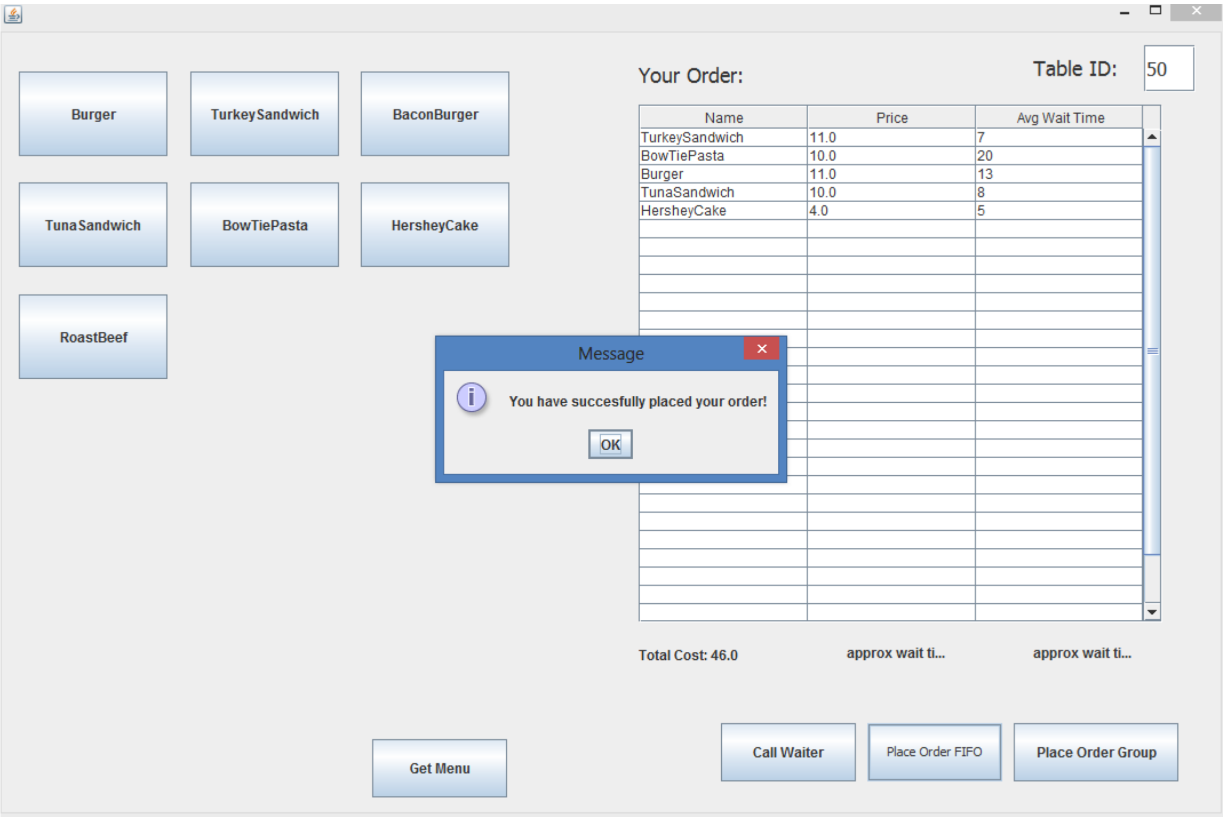


Table ID: 50

Customer GUI



Your Order:

Table ID: 50

| Name | Price | Avg Wait Time |
|---|---|---|
| TurkeySandwich | 11.0 | 7 |
| BowTiePasta | 10.0 | 20 |
| Burger | 11.0 | 13 |
| TunaSandwich | 10.0 | 8 |
| HersheyCake | 4.0 | 5 |

Total Cost: 46.0

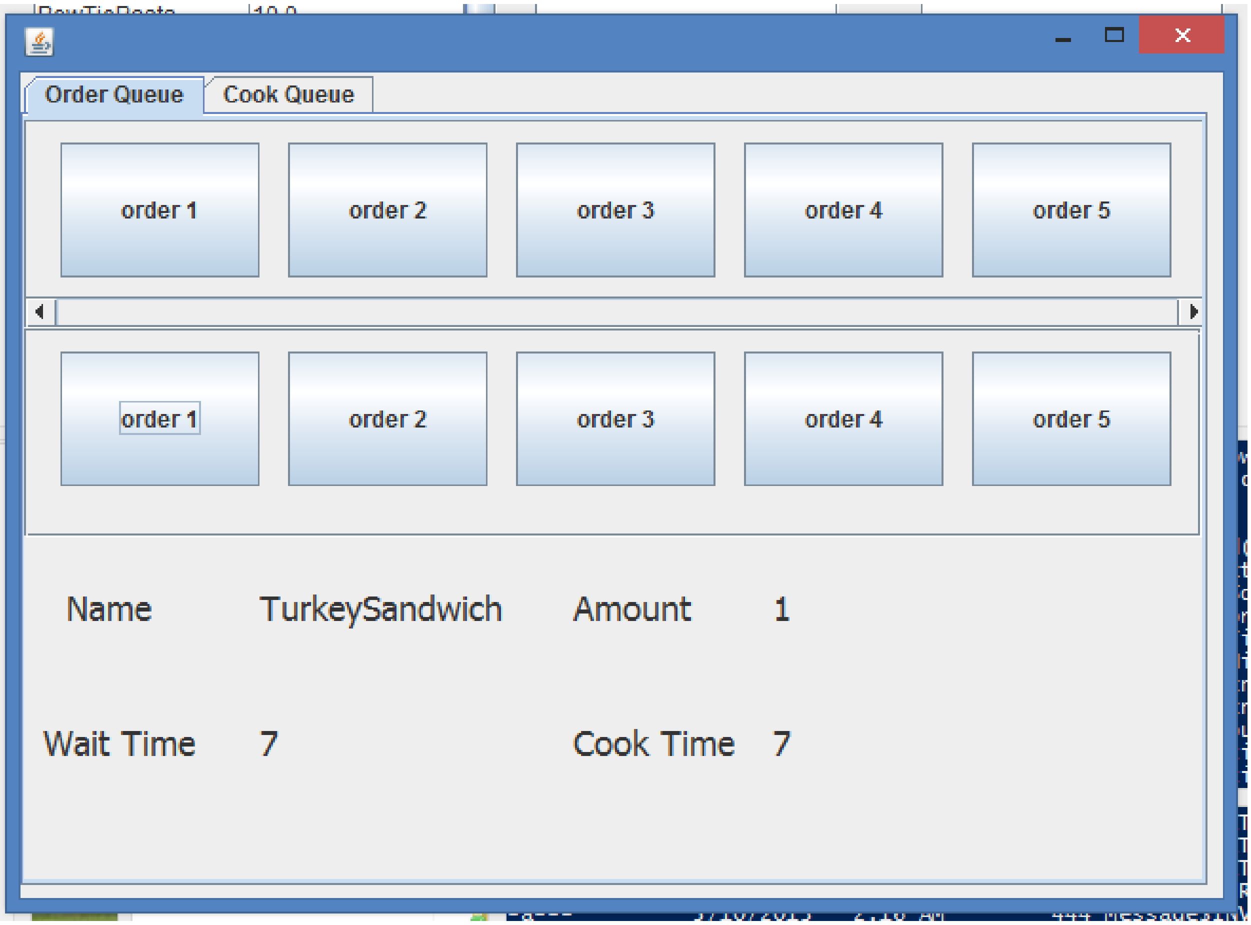
approx wait ti...

approx wait ti...

Message

You have succesfully placed your order!

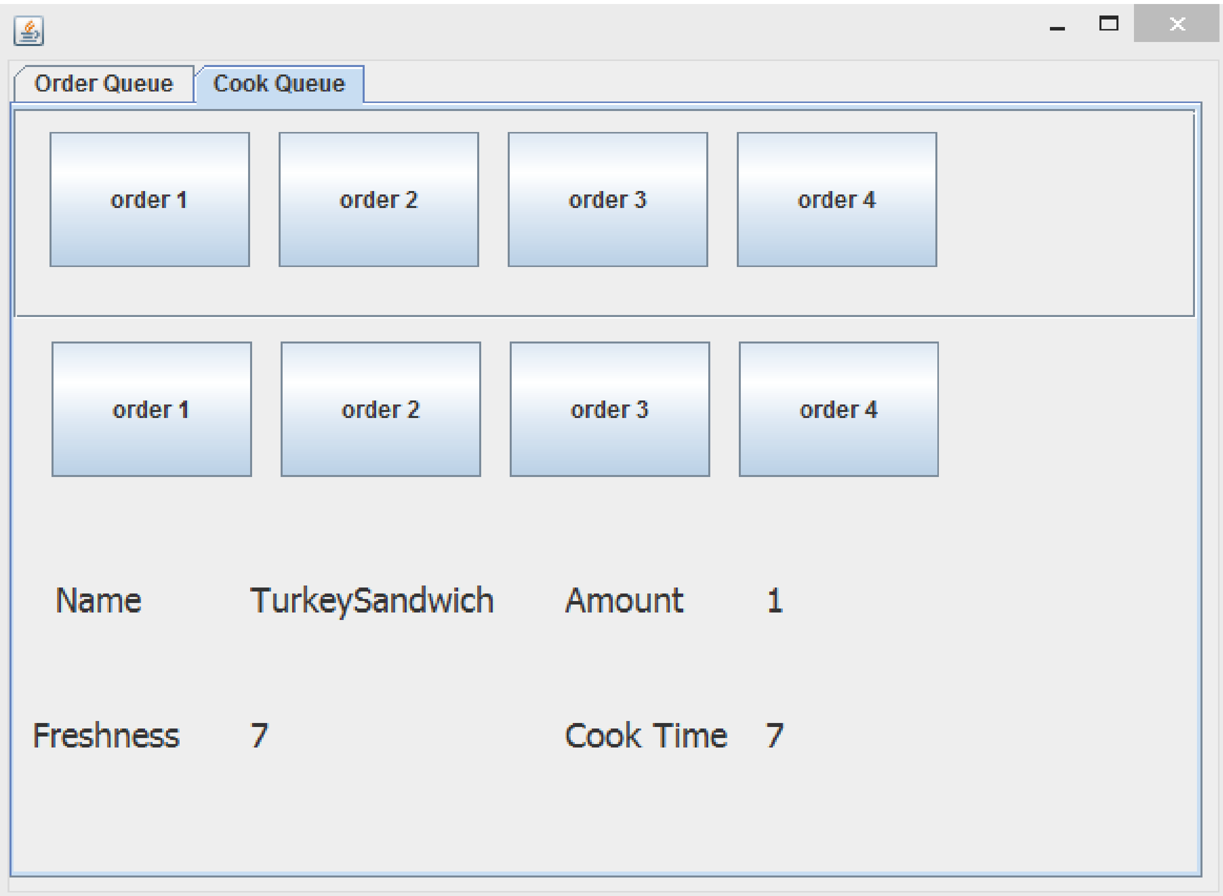
Customer GUI – Shown with Orders Being Placed



Name TurkeySandwich Amount 1

Wait Time 7 Cook Time 7

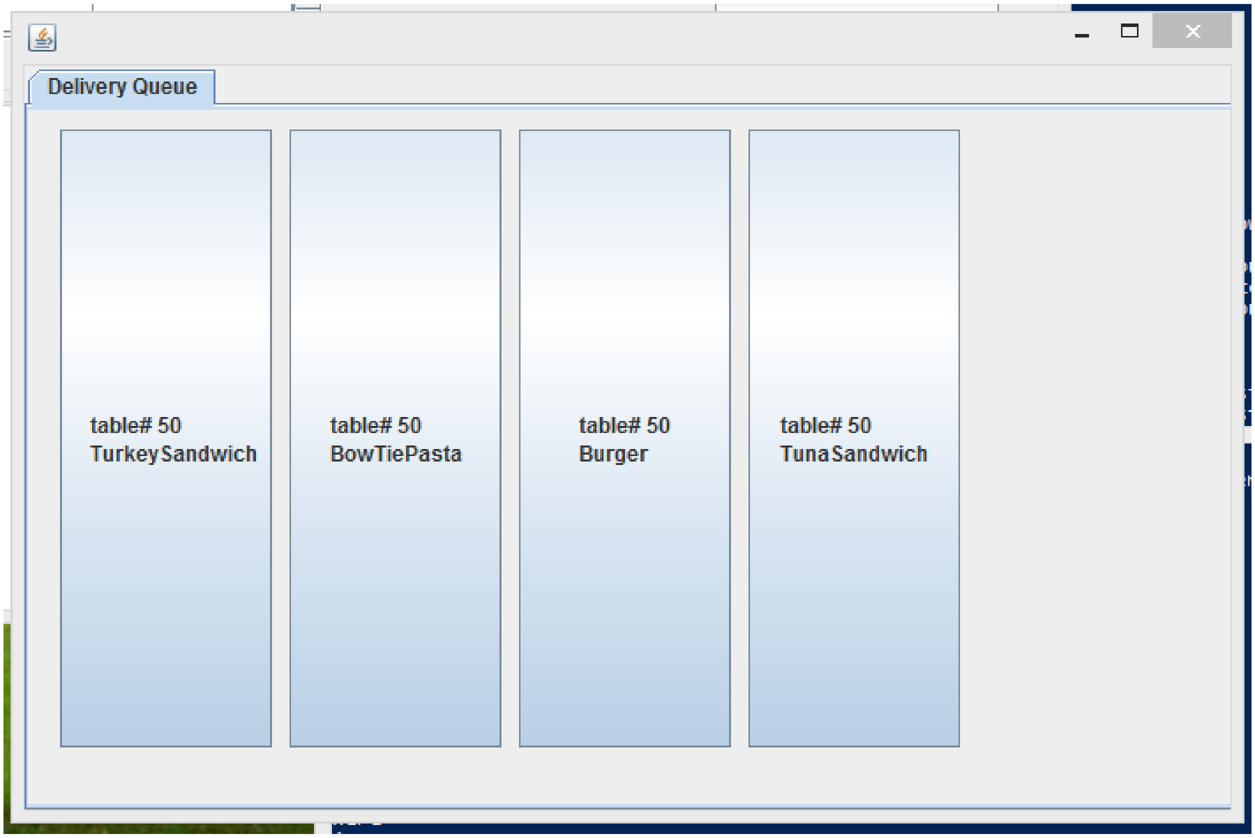
Chef GUI's Order Queue with Orders from the Customer



Name TurkeySandwich Amount 1

Freshness 7 Cook Time 7

Chef GUI showing the Current Orders Being Prepared



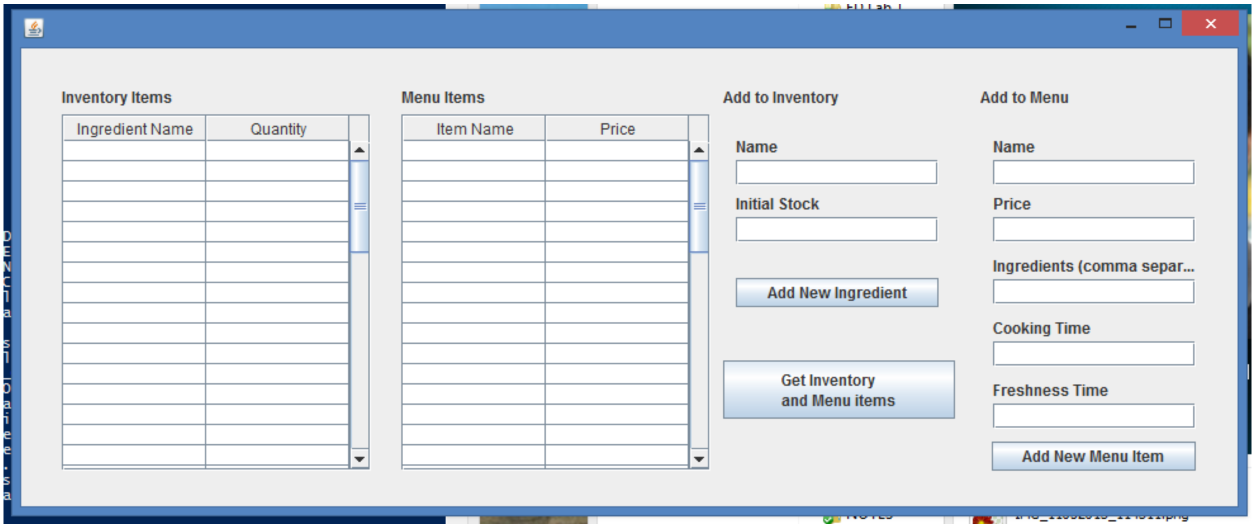
table# 50
TurkeySandwich

table# 50
BowTiePasta

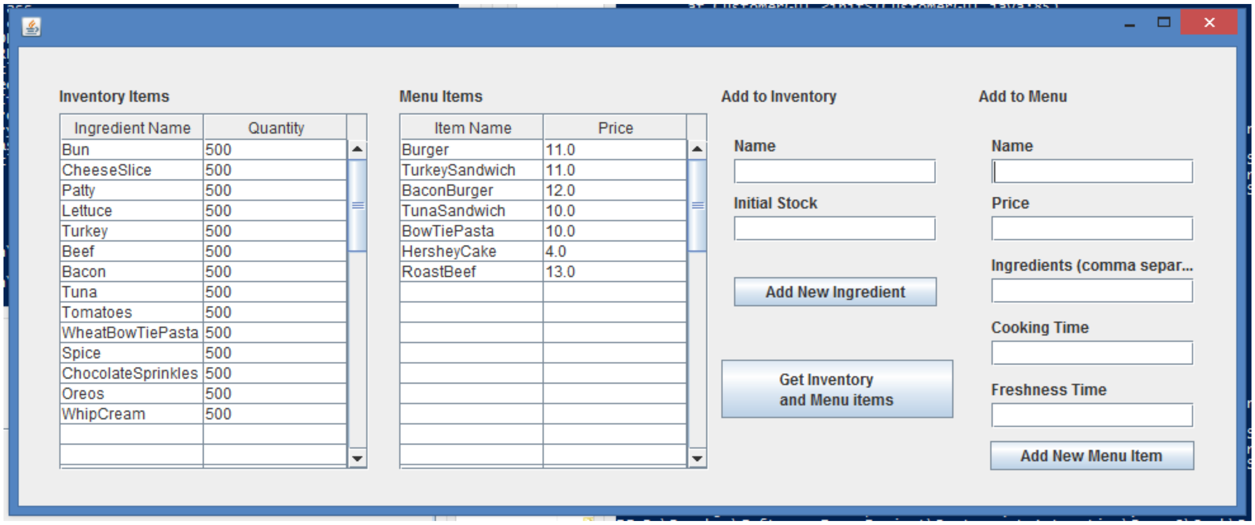
table# 50
Burger

table# 50
TunaSandwich

Waiter GUI showing Orders to be Delivered



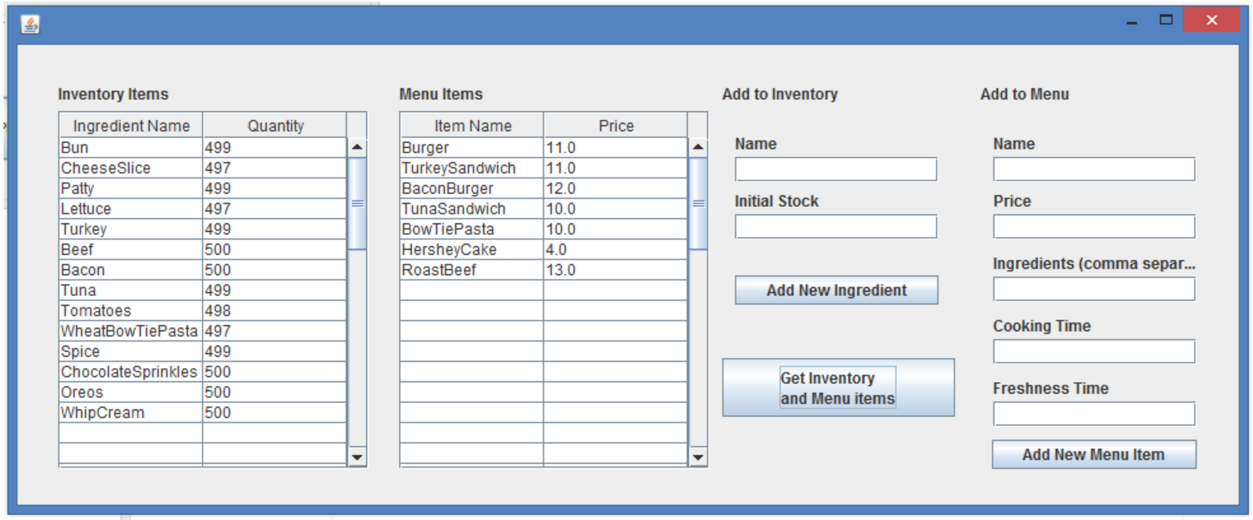
Manager GUI



| Ingredient Name | Quantity |
|---|---|
| Bun | 500 |
| CheeseSlice | 500 |
| Patty | 500 |
| Lettuce | 500 |
| Turkey | 500 |
| Beef | 500 |
| Bacon | 500 |
| Tuna | 500 |
| Tomatoes | 500 |
| WheatBowTiePasta | 500 |
| Spice | 500 |
| ChocolateSprinkles | 500 |
| Oreos | 500 |
| WhipCream | 500 |

| Item Name | Price |
|---|---|
| Burger | 11.0 |
| TurkeySandwich | 11.0 |
| BaconBurger | 12.0 |
| TunaSandwich | 10.0 |
| BowTiePasta | 10.0 |
| HersheyCake | 4.0 |
| RoastBeef | 13.0 |

Manager GUI with List of Ingredients in the Inventory



| Ingredient Name | Quantity |
|---|---|
| Bun | 499 |
| CheeseSlice | 497 |
| Patty | 499 |
| Lettuce | 497 |
| Turkey | 499 |
| Beef | 500 |
| Bacon | 500 |
| Tuna | 499 |
| Tomatoes | 498 |
| WheatBowTiePasta | 497 |
| Spice | 499 |
| ChocolateSprinkles | 500 |
| Oreos | 500 |
| WhipCream | 500 |

| Item Name | Price |
|---|---|
| Burger | 11.0 |
| TurkeySandwich | 11.0 |
| BaconBurger | 12.0 |
| TunaSandwich | 10.0 |
| BowTiePasta | 10.0 |
| HersheyCake | 4.0 |
| RoastBeef | 13.0 |

Manager GUI with List of Updated Ingredients in the Inventory

# 11. Design of Tests

## 11.1 Unit Test Cases

### 11.1.1 Customer

<table>
<tr><td colspan="2"><b>Test-case Identifier:</b> TC - 1<br><b>Function Tested:</b> Customer::sendOrder (MenuOrder order) : Boolean throws exception<br><b>Pass/Fail Criteria:</b> The test passes if menu order is successfully sent to the controller.</td></tr>
<tr><td><b>Test procedure</b></td><td><b>Expected Results</b></td></tr>
</table>

| | |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed, Controller receives the menu order, returns true whether order can be placed |
| -Call Function (Fail) | -function returns false if order cannot be placed (ingredients low, etc.)<br>- message fails to be sent, function throws exception |

<table>
<tr><td colspan="2"><b>Test-case Identifier:</b> TC - 2<br><b>Function Tested:</b> Customer::requestAssist () : Boolean throws exception<br><b>Pass/Fail Criteria:</b> The test passes if the customer successfully able to send a message to the controller indicating that he needs assistance.</td></tr>
<tr><td><b>Test procedure</b></td><td><b>Expected Results</b></td></tr>
<tr><td>-Call Function (Pass)</td><td>-Correct data to be sent is passed, Controller receives the message, returns true whether the waiter has received the request</td></tr>
<tr><td>-Call Function (Fail)</td><td>Waiter does not receive the request, returns false.<br>- message fails to be sent, function throws exception</td></tr>
</table>

<table>
<tr><td colspan="2"><b>Test-case Identifier:</b> TC - 3<br><b>Function Tested:</b> Customer::viewWaitTime (Menuitem m) : int throws exception<br><b>Pass/Fail Criteria:</b> The test passes if the correct wait time in seconds is returned from the controller for the menu item passed as a parameter.</td></tr>
<tr><td><b>Test procedure</b></td><td><b>Expected Results</b></td></tr>
<tr><td>-Call Function (Pass)</td><td>-Correct data to be sent is passed, Controller receives the message, returns the number of seconds for the menu item that has been already ordered.</td></tr>
<tr><td>-Call Function (Fail)</td><td>Otherwise will return -1 if the items has not been ordered yet.<br>- message fails to be sent, function throws exception</td></tr>
</table>

<table>
<tr><td colspan="2"><b>Test-case Identifier:</b> TC-4<br><b>Function Tested:</b> Customer::Rate (Menuitem m, int rating, string comment) : Boolean throws exception<br><b>Pass/Fail Criteria:</b> The test passes if the controller receives the rating for the menu item and returns true if the rating was applied, false if not.</td></tr>
<tr><td><b>Test procedure</b></td><td><b>Expected Results</b></td></tr>
</table>

| | |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed, Controller receives the message, returns true if the rating was successful |
| -Call Function (Fail) | -controller determines rating cannot be applied due to incorrect arguments, and function returns false.<br>- message fails to be sent, function throws exception |

<table>
<tr><td colspan="2"><b>Test-case Identifier:</b> TC - 5<br><b>Function Tested:</b> Customer::ViewMenu() : Menu throws exception<br><b>Pass/Fail Criteria:</b> The test passes if a Menu object is returned populated with all the menu items on the menu.</td></tr>
<tr><td><b>Test procedure</b></td><td><b>Expected Results</b></td></tr>
<tr><td>-Call Function (Pass)</td><td>-Correct data to be sent is passed, Controller receives the message, returns the menu object.</td></tr>
<tr><td>-Call Function (Fail)</td><td>Otherwise will return null, indicating a serious error has occurred or a menu has not been set.<br>- message fails to be sent, function throws exception</td></tr>
</table>

<table>
<tr><td colspan="2"><b>Test-case Identifier:</b> TC- 6<br><b>Function Tested:</b> Customer::CancelOrder (TableOrder o) : Boolean throws exception<br><b>Pass/Fail Criteria:</b> The test passes if the table order is successfully removed from the chef’s queue via the controller.</td></tr>
<tr><td><b>Test procedure</b></td><td><b>Expected Results</b></td></tr>
<tr><td>-Call Function (Pass)</td><td>-Correct data to be sent is passed, Controller receives the message, returns true on successful cancelations</td></tr>
<tr><td>-Call Function (Fail)</td><td>Function returns false if the order cannot be cancelled.<br>- message fails to be sent, function throws exception</td></tr>
</table>

### 11.1.2 Menu

**Test-case Identifier:** TC - 8
**Function Tested:** Menu::AddItem (Menuitem m) : Boolean
**Pass/Fail Criteria:** The test passes the menu item passed is added to the menu.

| Test procedure | Expected Results |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed, item is added to the menu. |
| -Call Function (Fail) | - Item cannot be added returns false. |

**Test-case Identifier:** TC - 9
**Function Tested:** Menu::RemoveItem (Menuitem m) : Boolean
**Pass/Fail Criteria:** The test passes if the menu item passed is removed from the menu.

| Test procedure | Expected Results |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed,<br>Item is removed from the menu. |
| -Call Function (Fail) | - Item fails to be removed, function returns false. |

**Test-case Identifier:** TC - 10
**Function Tested:** Menu::ViewItem (String name) : MenuItem M
**Pass/Fail Criteria:** The test passes if a menu item object is returned with the corresponding name.

| Test procedure | Expected Results |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed,<br>Function returns a menu item corresponding to the name passed. |
| -Call Function (Fail) | - menu item does not exists, function fails and returns null. |

<table>
<tr><td colspan="2"><b>Test-case Identifier:</b> TC - 11<br><b>Function Tested:</b> Menu::DisableItem (Menuitem m) : Boolean<br><b>Pass/Fail Criteria:</b> The test passes the items pass in as an argument is disabled from the menu.</td></tr>
<tr><td><b>Test procedure</b></td><td><b>Expected Results</b></td></tr>
<tr><td>-Call Function (Pass)</td><td>-Correct data to be sent is passed, function returns true of item is disabled from the menu.</td></tr>
<tr><td>-Call Function (Fail)</td><td>- item cannot be disabled, function returns false</td></tr>
</table>

<table>
<tr><td colspan="2"><b>Test-case Identifier:</b> TC - 12<br><b>Function Tested:</b> Menu::ViewMenu () : ArrayList<MenuItem><br><b>Pass/Fail Criteria:</b> The test passes if the list of menu items associated with the menu is returned.</td></tr>
<tr><td><b>Test procedure</b></td><td><b>Expected Results</b></td></tr>
<tr><td>-Call Function (Pass)</td><td>-Correct data to be sent is passed, list of menu items is returned.</td></tr>
<tr><td>-Call Function (Fail)</td><td>- List cannot be returned or is nonexistent, function returns null.</td></tr>
</table>

<table>
<tr><td colspan="2"><b>Test-case Identifier:</b> TC - 13<br><b>Function Tested:</b> Menu::ViewRatedList () : ArrayList<Rating><br><b>Pass/Fail Criteria:</b> The test passes if the correct wait time in seconds is returned from the controller for the menu item passed as a parameter.</td></tr>
<tr><td><b>Test procedure</b></td><td><b>Expected Results</b></td></tr>
<tr><td>-Call Function (Pass)</td><td>-Correct data to be sent is passed, list of ratings for the menu is returned.</td></tr>
<tr><td>-Call Function (Fail)</td><td>- List cannot be returned or there are no ratings for the menu, function returns null.</td></tr>
</table>

### 11.1.3 Chef

**Test-case Identifier:** TC - 14
**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 - 15
**Function Tested:** Chef::DeleteItem (Menuitem m) : Boolean
**Pass/Fail Criteria:** The test passes if the menu item passed as an argument is removed from the chef queue.

| **Test procedure** | **Expected Results** |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed,<br>Function returns true if the menu item is removed from the chef queue. |
| -Call Function (Fail) | - Function returns false, of the item cannot be removed or cannot be found. |

**Test-case Identifier:** TC - 16
**Function Tested:** Chef::NotifyCatastrophe () : Boolean throws exception
**Pass/Fail Criteria:** The test passes if the controller receives the message for a catastrophe and function returns true.

| **Test procedure** | **Expected Results** |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed, functions returns true when controller acknowledges message. |
| -Call Function (Fail) | Controllers does not acknowledge catastrophe.<br>- message fails to be sent, function returns false, |

<table>
<tr><td colspan="2"><b>Test-case Identifier:</b> TC - 17<br><b>Function Tested:</b> Chef::FinishedItem (MenuItem m) : Boolean throws exception<br><b>Pass/Fail Criteria:</b> The test passes if the item is removed from temporary storage and passed to the waiter via the controller.</td></tr>
<tr><td><b>Test procedure</b></td><td><b>Expected Results</b></td></tr>
<tr><td>-Call Function (Pass)<br>-Call Function (Fail)</td><td>-Correct data to be sent is passed, function returns true if the controller successfully passes the MenuItem to the waiter to be delivered.<br>MenuItem incorrect or controller error, function returns false.<br>- message fails to be sent, function throws exception</td></tr>
</table>

<table>
<tr><td colspan="2"><b>Test-case Identifier:</b> TC - 18<br><b>Function Tested:</b> Chef::AddItem (Menuitem m) : Boolean<br><b>Pass/Fail Criteria:</b> The test passes if the menu item that is passed is successfully scheduled into the chef’s queue.</td></tr>
<tr><td><b>Test procedure</b></td><td><b>Expected Results</b></td></tr>
<tr><td>-Call Function (Pass)<br>-Call Function (Fail)</td><td>-Correct data to be sent is passed, function returns true if menu item is scheduled.<br>Function returns false, if menu item is incorrect or cannot be scheduled.</td></tr>
</table>

<table>
<tr><td colspan="2"><b>Test-case Identifier:</b> TC - 19<br><b>Function Tested:</b> Chef::ViewQueue () : ArrayList&lt;MenuItem&gt;<br><b>Pass/Fail Criteria:</b> The test passes if the list of menu items that are currently in the queue are returned.</td></tr>
<tr><td><b>Test procedure</b></td><td><b>Expected Results</b></td></tr>
<tr><td>-Call Function (Pass)<br>-Call Function (Fail)</td><td>-Correct data to be sent is passed, function returns the list of menu items.<br>List does not exist, function returns null.</td></tr>
</table>

### 11.1.4 Waiter

**Test-case Identifier:** TC - 20
**Function Tested:** Waiter::AddItem (MenuItem m) : Boolean
**Pass/Fail Criteria:** The test passes if menu item is added to the ready queue or added to a table order currently being waited on.

| **Test procedure** | **Expected Results** |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed, menu items is added to the ready queue or added to a table order that is waiting, function returns true. |
| -Call Function (Fail) | Menu item cannot be added or is improper format, function returns false. |

**Test-case Identifier:** TC - 21
**Function Tested:** Waiter::DeleteItem (Menuitem m) : Boolean
**Pass/Fail Criteria:** The test passes if the menu item passed is removed from the ready queue.

| **Test procedure** | **Expected Results** |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed, menu item is removed from the waiter's ready queue, function returns true. |
| -Call Function (Fail) | - item cannot be removed, or does note exists or improper format, function returns false. |

**Test-case Identifier:** TC - 22
**Function Tested:** Waiter::ViewQueue () : ArrayList<MenuItem>
**Pass/Fail Criteria:** The test passes if the list of menu items that are currently in the waiter's queue are returned.

| **Test procedure** | **Expected Results** |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed, function returns list of menu items in the waiter's queue. |
| -Call Function (Fail) | - The list of items are empty or failed to be retrieved, function returns null. |

**Test-case Identifier:** TC - 23
**Function Tested:** Waiter::Notify (int tableNum) : Boolean throws exception
**Pass/Fail Criteria:** The test passes if the table with the table number passed as a parameter is notified that a waiter is on the way.

| Test procedure | Expected Results |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed, function returns true if the customer at table tableNum is notified via the controller. |
| -Call Function (Fail) | Function returns false if tableNum does not exists or did not request assistance from waiter.<br>- message fails to be sent, function throws exception |

### 11.1.5 Manager

**Test-case Identifier:** TC - 24
**Function Tested:** Manager::ViewInventory () : ArrayList<InventoryItem> throws exception
**Pass/Fail Criteria:** The test passes if the function returns the list of inventory items.

| Test procedure | Expected Results |
|---|---|
| -Call Function (Pass) | -function returns list of inventory items . |
| -Call Function (Fail) | -function returns null if there are not inventory items.<br>- message fails to be sent, function throws exception |

<table>
<tr><td colspan="2"><b>Test-case Identifier:</b> TC – 25<br><b>Function Tested:</b> Manager::AddInventoryitem (InventoryItem i) : Boolean throws exception<br><b>Pass/Fail Criteria:</b> The test passes if the InventoryItem passes is successfully added to the inventory.</td></tr>
<tr><td><b>Test procedure</b></td><td><b>Expected Results</b></td></tr>
<tr><td>-Call Function (Pass)<br>-Call Function (Fail)</td><td>-Correct data to be sent is passed, function returns true if item is added.<br>Function returns false if item cannot be added or improper data is passed.<br>- message fails to be sent, function throws exception</td></tr>
</table>

<table>
<tr><td colspan="2"><b>Test-case Identifier:</b> TC - 26<br><b>Function Tested:</b> Manager::RemoveInventoryItem (InventoryItem i) : Boolean throws exception<br><b>Pass/Fail Criteria:</b> The test passes if the function successfully removes the inventory item passes as an argument.</td></tr>
<tr><td><b>Test procedure</b></td><td><b>Expected Results</b></td></tr>
<tr><td>-Call Function (Pass)<br>-Call Function (Fail)</td><td>-Correct data to be sent is passed, function returns true if Inventory Item is removed.<br>- Function returns false if item could not be removed or improper item was passed.<br>- message fails to be sent, function throws exception</td></tr>
</table>

**Test-case Identifier:** TC - 27
**Function Tested:** Manager::EditInventoryItem (InvetoryItem) : Boolean throws exception
**Pass/Fail Criteria:** The test passes if the inventory item passes replaces the one currently in the inventory.

| Test procedure | Expected Results |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed, function returns true after inventory item is successfully replaced. |
| -Call Function (Fail) | Function returns false if improper data passed or could not edit the item.<br>- message fails to be sent, function throws exception |

**Test-case Identifier:** TC - 28
**Function Tested:** Manager::ViewPopularity () : ArrayList<Rating> throws exception
**Pass/Fail Criteria:** The test passes if the function returns a list of ratings for the menu.

| Test procedure | Expected Results |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed, function returns the list of ratings for the current menu. |
| -Call Function (Fail) | Function returns null of there are no ratings in the system.<br>- message fails to be sent, function throws exception |

### 11.1.6 Controller

**Test-case Identifier:** TC - 28
**Function Tested:** Controller::CommandHandler::SpawnThread(Message m) :Boolean
**Pass/Fail Criteria:** The test passes if the function successfully spawns a new thread which can handle the message passed in as an argument.

| **Test procedure** | **Expected Results** |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed, starts a new thread which will handle the message. |
| -Call Function (Fail) | Function returns false if passed message is not proper<br>- message fails to be sent, function throws exception |

**Test-case Identifier:** TC - 29
**Function Tested:** Controller::CommandHandler::handleConn(Socket) : void
**Pass/Fail Criteria:** The test passes if the function successfully accepts a connection on the socket passed in.

| **Test procedure** | **Expected Results** |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed, function will block until a connection has been accepter |
| -Call Function (Fail) | Function returns without accepting the connection on the socket. |

**Test-case Identifier:** TC - 30
**Function Tested:** Controller::CommandHandler::SendToHandler(Message m) : Boolean throws exception
**Pass/Fail Criteria:** The test passes if the function successfully sends the message to the appropriate handler.

| Test procedure | Expected Results |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed, function returns true when handler has received the message. |
| -Call Function (Fail) | Function returns false if message is improper or handler has not received the message.<br>- message fails to be sent, function throws exception |

**Test-case Identifier:** TC - 31
**Function Tested:** Controller::CommandHandler::init(): Boolean
**Pass/Fail Criteria:** The test passes if the function initializes the command handler.

| Test procedure | Expected Results |
|---|---|
| -Call Function (Pass) | -function returns true on successful initialization. |
| -Call Function (Fail) | Function returns false if an error occurred during initialization. |

**Test-case Identifier:** TC - 32
**Function Tested:** Controller::CommandHandler::ShutDown : void
**Pass/Fail Criteria:** The test passes If the command handler successfully shuts down.

| Test procedure | Expected Results |
|---|---|
| -Call Function (Pass) | -function shuts down the command handler. |
| -Call Function (Fail) | - command handler fails to shutdown |

**Test-case Identifier:** TC - 33

**Function Tested:** Controller::InventoryHandler::getInventoryInfo() : Inventory

**Pass/Fail Criteria:** The test passes if the function returns an inventory object updated with the correct inventory information.

| **Test procedure** | **Expected Results** |
|---|---|
| -Call Function (Pass) | - Function returns correct inventory object. |
| -Call Function (Fail) | - function returns null if inventory does not exist, |

**Test-case Identifier:** TC - 34

**Function Tested:** Controller::InventoryHandler::updateInventory (Message m) : Boolean throws exception

**Pass/Fail Criteria:** The test passes if the function correctly updates the inventory with the passed object.

| **Test procedure** | **Expected Results** |
|---|---|
| -Call Function (Pass) | -correct data is passed, function returns true if inventory is successfully updated. |
| -Call Function (Fail) | Function returns false, if inventory cannot be updated or arguments is improper.<br>- message fails to be sent, function throws exception |

**Test-case Identifier:** TC - 35

**Function Tested:** Controller::InventoryHandler::isLow(String item) : Boolean

**Pass/Fail Criteria:** The test passes if the function successfully returns true or false depending on if the item is low

| **Test procedure** | **Expected Results** |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed, function returns true or false after determining if the item is low. |
| -Call Function (Fail) | Function returns null or an incorrect value if the passed item was not found. |

**Test-case Identifier:** TC - 36

**Function Tested:** Controller::InventoryHandler::deduct(String name) : boolean
**Pass/Fail Criteria:** The test passes if the function successfully deducts one from the count of the number of items in the inventory with an item matching name.

| Test procedure | Expected Results |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed, function returns true after deducting one from the count of the passed in item |
| -Call Function (Fail) | Function returns null or false if it cannot find the item or cannot access the inventory. |

**Test-case Identifier:** TC - 37
**Function Tested:** Controller::NotificationHandler::handleMessage(String message) : Boolean throws exception
**Pass/Fail Criteria:** The test passes if the function successfully sends the message to the appropriate subscribers of the message.

| Test procedure | Expected Results |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed, function returns true after sending the message to the appropriate subscribers. |
| -Call Function (Fail) | Function returns false if an improper message was passed.<br>- message fails to be sent, function throws exception |

**Test-case Identifier:** TC - 38
**Function Tested:** Controller::NotificationHandler:: notify(String notification, Socket connection) : Boolean throws exception
**Pass/Fail Criteria:** The test passes if the function successfully sends the notification to the socket passed in.

| Test procedure | Expected Results |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed, function returns true after sending notification over socket |
| -Call Function (Fail) | Function returns false notification or socket is improper, |

| | |
|---|---|
| | - message fails to be sent, function throws exception |

**Test-case Identifier:** TC - 40
**Function Tested:** Controller::LogHandler::writeLog(String s) : Boolean
**Pass/Fail Criteria:** The test passes if the function successfully writes the log passed in as an argument.

| Test procedure | Expected Results |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed, function returns true after writing to the log. |
| -Call Function (Fail) | Function returns false if it cannot write to the log. |

**Test-case Identifier:** TC - 41
**Function Tested:** Controller::LogHandler::readLogs() : List<String>
**Pass/Fail Criteria:** The test passes if the function successfully returns a list of logs.

| Test procedure | Expected Results |
|---|---|
| -Call Function (Pass) | -function returns list of logs. |
| -Call Function (Fail) | Function returns null of there are no logs in the system. |

**Test-case Identifier:** TC - 42
**Function Tested:** Controller::OrderHandler::addOrder(TableOrder order) : Boolean
**Pass/Fail Criteria:** The test passes if the function successfully adds the table order passed.

| Test procedure | Expected Results |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed, function returns true after adding the table order. |
| -Call Function (Fail) | Function returns false if table order is improper or cannot be added. |

**Test-case Identifier:** TC - 43
**Function Tested:** Controller::OrderHandler::editOrder(int orderNum, TableOrder o) : Boolean

**Pass/Fail Criteria:** The test passes if the function successfully updates the order with id, orderNum with the table order passed.

| **Test procedure** | **Expected Results** |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed, function returns true after successfully updates the order. |
| -Call Function (Fail) | Function returns false if order cannot be updated, improper order passed, or cannot id cannot be found. |

**Test-case Identifier:** TC - 44
**Function Tested:** Controller::OrderHandler::getWaittime(int orderNum) : int
**Pass/Fail Criteria:** The test passes if the function successfully return the wait time for the order.

| **Test procedure** | **Expected Results** |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed, function correctly returns the wait time for the order. |
| -Call Function (Fail) | Function returns -1 if order cannot be found. |

**Test-case Identifier:** TC - 45
**Function Tested:** Controller::OrderHandler::flagItemDone(MenuItem item) Boolean throws exception
**Pass/Fail Criteria:** The test passes if the function successfully signals the waiter to add the item to be delivered.

| **Test procedure** | **Expected Results** |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed, function returns true after sending item to the waiter. |
| -Call Function (Fail) | Function returns false if item is improper.<br>- message fails to be sent, function throws exception |

**Test-case Identifier:** TC - 46
**Function Tested:** Controller::DatabaseHandler::getInventory() : Menu
**Pass/Fail Criteria:** The test passes if the function successfully returns the menu updated

with the current inventory.

| Test procedure | Expected Results |
|---|---|
| -Call Function (Pass) | -function returns the menu updated with the current inventory. |
| -Call Function (Fail) | Function returns null if there is no inventory. |

**Test-case Identifier:** TC - 47
**Function Tested:** Controller::DatabaseHandler::addNewInventory(String name, ...) : Boolean
**Pass/Fail Criteria:** The test passes if the function successfully adds a new item to the inventory

| Test procedure | Expected Results |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed, function returns true after adding a new item to the inventory. |
| -Call Function (Fail) | Function returns false if item cannot be added. |

**Test-case Identifier:** TC - 48
**Function Tested:** Controller::DatabaseHandler:: removeFromInventory(String name):Boolean
**Pass/Fail Criteria:** The test passes if the function successfully remove the item matching the name passed.

| Test procedure | Expected Results |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed, function returns true after the item associated with the name is removed. |
| -Call Function (Fail) | Function returns false if it cannot find the item. |

**Test-case Identifier:** TC - 49
**Function Tested:** Controller::DatabaseHandler:: writeLog(String log) : Boolean
**Pass/Fail Criteria:** The test passes if the function successfully writes a log string to the log.

| Test procedure | Expected Results |
|---|---|

| | |
|---|---|
| -Call Function (Pass)<br>-Call Function (Fail) | -Correct data to be sent is passed, function returns true after writing a string to the log.<br>Function returns false if it cannot add a string to the log, or string pass in is improper. |

<table>
<tr><td colspan="2"><b>Test-case Identifier:</b> TC - 50<br><b>Function Tested:</b> Controller::DatabaseHandler:: getPrevLogs():List<String><br><b>Pass/Fail Criteria:</b> The test passes if the function successfully returns the list of logs previously recorded</td></tr>
<tr><td><b>Test procedure</b></td><td><b>Expected Results</b></td></tr>
<tr><td>-Call Function (Pass)<br>-Call Function (Fail)</td><td>-function returns the list of previous logs.<br>Function returns null of there are no previous logs.</td></tr>
</table>

<table>
<tr><td colspan="2"><b>Test-case Identifier:</b> TC - 51<br><b>Function Tested:</b> Controller::DatabaseHandler:: updateInventory(String name, int num) : Boolean<br><b>Pass/Fail Criteria:</b> The test passes if the function successfully updates inventory in the database.</td></tr>
<tr><td><b>Test procedure</b></td><td><b>Expected Results</b></td></tr>
<tr><td>-Call Function (Pass)<br>-Call Function (Fail)</td><td>-Correct data to be sent is passed, function returns true after updating the database.<br>Function returns false, if data passed is improper or cannot be updated .</td></tr>
</table>

<table>
<tr><td><b>Test-case Identifier:</b> TC - 52<br><b>Function Tested:</b> Controller::DatabaseHandler:: removeOneInventory(String name) :</td></tr>
</table>

<table>
<tr><td colspan="2">Boolean<br><b>Pass/Fail Criteria:</b> The test passes if the function successfully removes one from the item passed in</td></tr>
<tr><td><b>Test procedure</b></td><td><b>Expected Results</b></td></tr>
<tr><td>-Call Function (Pass)</td><td>-Correct data to be sent is passed, function returns true after removing one from the inventory of the item passed.</td></tr>
<tr><td>-Call Function (Fail)</td><td>Function returns false if the argument is improper or cannot be found.</td></tr>
</table>

<table>
<tr><td colspan="2"><b>Test-case Identifier:</b> TC - 53<br><b>Function Tested:</b> Controller::MenuHandler::init() : boolean<br><b>Pass/Fail Criteria:</b> The test passes if the function successfully initializes the menu handler</td></tr>
<tr><td><b>Test procedure</b></td><td><b>Expected Results</b></td></tr>
<tr><td>-Call Function (Pass)</td><td>-function returns true after initializing the menu handler.</td></tr>
<tr><td>-Call Function (Fail)</td><td>Function returns false if it cannot initialize the menu handler.</td></tr>
</table>

<table>
<tr><td colspan="2"><b>Test-case Identifier:</b> TC - 54<br><b>Function Tested:</b> Controller::MenuHandler::handleMessage(Message m) :Boolean throws exception<br><b>Pass/Fail Criteria:</b> The test passes if the function successfully handles the message</td></tr>
<tr><td><b>Test procedure</b></td><td><b>Expected Results</b></td></tr>
<tr><td>-Call Function (Pass)</td><td>-Correct data to be sent is passed, function returns true after handling the message.</td></tr>
<tr><td>-Call Function (Fail)</td><td>Function returns false if it cannot handler the message.</td></tr>
</table>

<table>
<tr><td colspan="2"><b>Test-case Identifier:</b> TC - 55<br><b>Function Tested:</b> Controller::MenuHandler::AddMenuItem(MenuItem m) : Boolean throws exception<br><b>Pass/Fail Criteria:</b> The test passes if the function successfully adds the menu items passed</td></tr>
</table>

| to the menu. | |
|---|---|
| **Test procedure** | **Expected Results** |
| -Call Function (Pass) | -Correct data to be sent is passed, function returns true after adding the menu item. |
| -Call Function (Fail) | Function returns false if the menu item is improper. |

| **Test-case Identifier:** TC - 56<br>**Function Tested:** Controller::MenuHandler::RemoveMenuItem(MenuItem m) : Boolean<br>**Pass/Fail Criteria:** The test passes if the function successfully removes the menu item passed. | |
|---|---|
| **Test procedure** | **Expected Results** |
| -Call Function (Pass) | -Correct data to be sent is passed, function returns true after removing the menu item. |
| -Call Function (Fail) | Function returns false if it cannot remove the menu item, the menu item passed is improper, or it cannot find the menu item in the list. |

| **Test-case Identifier:** TC - 57<br>**Function Tested:** Controller::MenuHandler::getMenuItems(): ArrayList<MenuItem><br>**Pass/Fail Criteria:** The test passes if the function successfully returns the list of menu items that are part of the menu. | |
|---|---|
| **Test procedure** | **Expected Results** |
| -Call Function (Pass) | -Correct data to be sent is passed, function returns the list of menu items that are part of the menu. |
| -Call Function (Fail) | Function returns null if there are no menu items in the menu. |

| **Test-case Identifier:** TC - 58<br>**Function Tested:** Controller::MenuHandler::updateMenu(MenuItem m): Boolean<br>**Pass/Fail Criteria:** The test passes if the function successfully updates the old menu item |
|---|

| | |
|---|---|
| with the one passed in as a parameter. | |
| **Test procedure** | **Expected Results** |
| -Call Function (Pass) | -Correct data to be sent is passed, function returns true after updating the old menu item with the one passed in as an argument. |
| -Call Function (Fail) | Function returns false if it cannot find the menu item or the item passed is improper. . |

| | |
|---|---|
| **Test-case Identifier:** TC - 59<br>**Function Tested:** Controller::MenuHandler::DisableMenuItem(String name) : Boolean<br>**Pass/Fail Criteria:** The test passes if the function successfully disables the menu item passed in | |
| **Test procedure** | **Expected Results** |
| -Call Function (Pass) | -Correct data to be sent is passed, function returns true after disabling the menu items matching the name passed in. |
| -Call Function (Fail) | Function returns false if there<br>- message fails to be sent, function throws exception |

| | |
|---|---|
| **Test-case Identifier:** TC - 60<br>**Function Tested:** Controller::MenuHandler::getRatingList() : ArrayList<Rating><br>**Pass/Fail Criteria:** The test passes if the function successfully returns the list of ratings associated with the menu. | |
| **Test procedure** | **Expected Results** |
| -Call Function (Pass) | -Correct data to be sent is passed, function returns the list of ratings for the current menu. |
| -Call Function (Fail) | Function returns null of there are no ratings in the system. |

| |
|---|
| **Test-case Identifier:** TC - 61 |

**Function Tested:** Controller::MenuHandler::getRating(String name) : ArrayList<Rating>
**Pass/Fail Criteria:** The test passes if the function successfully returns the list of ratings associated with the menu item that matches the given name.

| **Test procedure** | **Expected Results** |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed, function returns the list of ratings for the menu item that matches the name. |
| -Call Function (Fail) | Function returns null if it cannot find the menu item. |

**Test-case Identifier:** TC - 62
**Function Tested:** Controller::MenuHandler::getMenuItems() : ArrayList<MenuItem>
**Pass/Fail Criteria:** The test passes if the function successfully returns the list of menu items associated with the menu.

| **Test procedure** | **Expected Results** |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed, function returns the list of menu items for this menu. |
| -Call Function (Fail) | Function returns null of there are no menu items in this menu |

**Test-case Identifier:** TC - 63
**Function Tested:** Controller::MenuHandler::setRating(String name, int rating, String comment) : Boolean
**Pass/Fail Criteria:** The test passes if the function successfully adds a rating for the menu item.

| **Test procedure** | **Expected Results** |
|---|---|
| -Call Function (Pass) | -Correct data to be sent is passed, function returns true after adding a rating for the menu item that matches the name passed in. |
| -Call Function (Fail) | Function returns false if the arguments are improper or it cannot find the menu item. |

## 11.2 Test Coverage

The test cases cover all the functionality of every class, however since the functions of classes will communicate and talk to other classes, the test coverage also ensures that testing is through for cross class communication and therefore covers the system as a whole. In essence, by testing the individual functions of every class, we cover the entire system as a whole. However, the way we test these functions are important for integration between the different classes which is discussed in the next section.

## 11.3 Integration Testing Strategy

Integration is highly important for our system since the functionality of the system as a whole depends on the interaction of each component of the system. Our approach to testing the integration of all the components is a bottom-up approach. We will first start with unit testing functions that belong to the classes and do not require communication with the network. For example, many of the classes contained in the controller are data processing classes and are created to improve the efficiency and robustness of the controller as a whole. For instance, the database handler and the logger are classes are just created to interact with the data base or log events. After testing these classes, we move on to testing communication between the classes. We start with testing the server communication of the controller, as it is the main communication of the system. We then move on to testing communication of pairs of controller-component.  This way, when a problem occurs it is easy to pinpoint it early on. If we were to take on a different approach, we would encounter problems that would have ambiguous roots. After testing pairs of components, we move on to events that require multiple components, and finish the unit tests that use these events.  For the initial parts of the testing we need to create test stubs for some of the input so that we can test edge cases that might not occur on a day to day basis and also to create data that needs to come in from different components before actually connecting the components. After testing all of the communication and using test stubs to test a variety of input, we will have a stable system.

## 11.4 Non-Functional Requirements Testing

To test the system's nonfunctional requirements, we will need to implement the system in an actual restaurant to see what functionality seemed to be better appealing visually or functionality that could be included that would increase the ease of use of the product. We would need to survey users of the product and ask questions of this sort: "What screens seemed hard to navigate?", "Chef, were you able to easily remove items from the queue?". Because the system is designed in such a way that upgrading is elementary, it will be easy to include these changes into an update that can be pushed to the system.

# 12. Project Management & History of Work

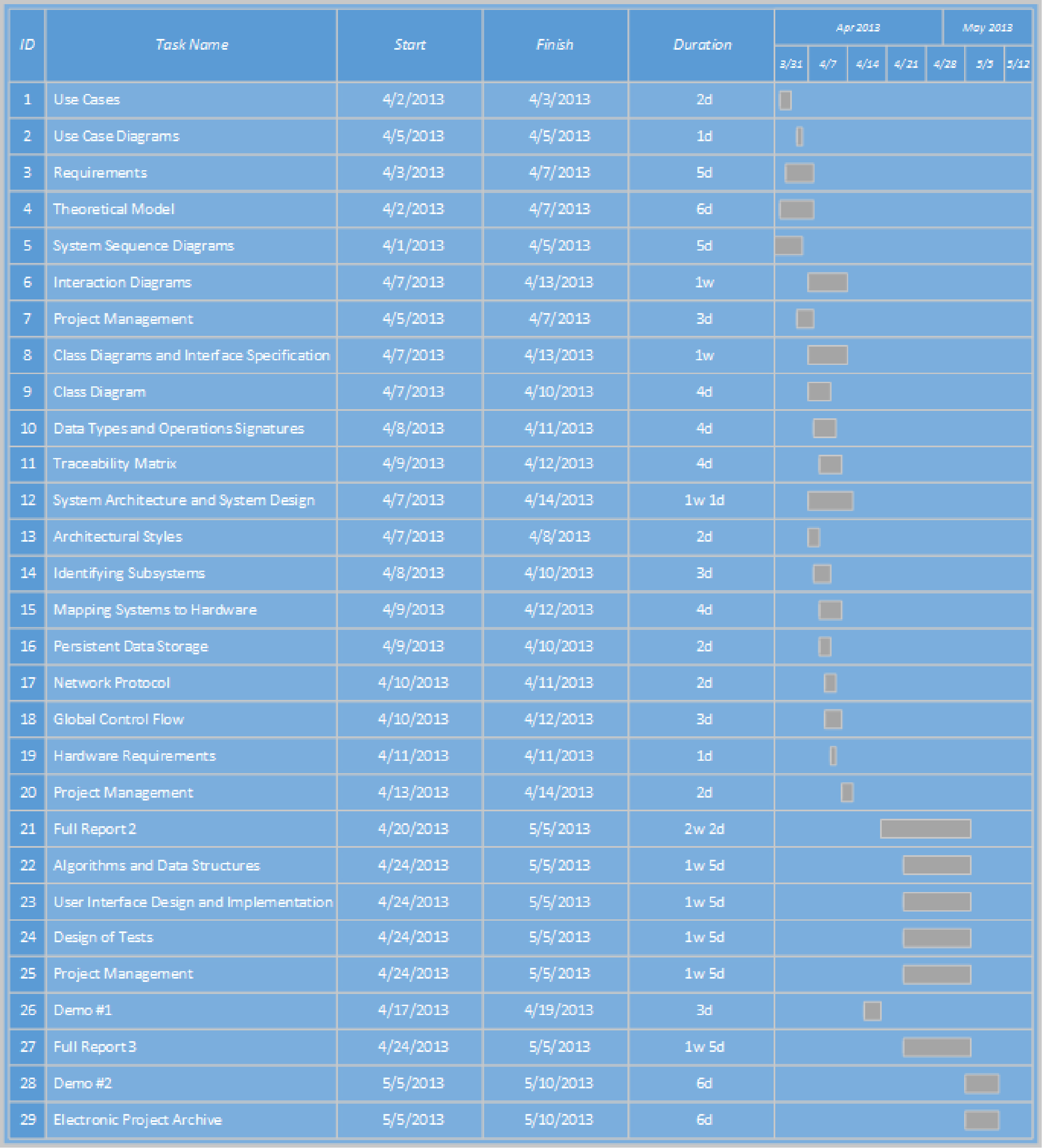
## 12.1 Merging Contributions from Individual Team Members

Compiling our group members' work for the reports was not too difficult.  We used SkyDrive to collaborate on one document, so we were able to edit and add our parts simultaneously.  We initially had a few issues with SkyDrive being buggy, but in the end, it resolved itself.

## 12.2 Project Coordination and Progress Report

For the current functionalities of the project, see the "Current Status" section below.  Our History of Work describes what we have done so far, as well as some problems we encountered, and how we adapted to these situations.

## 12.3 Plan of Work

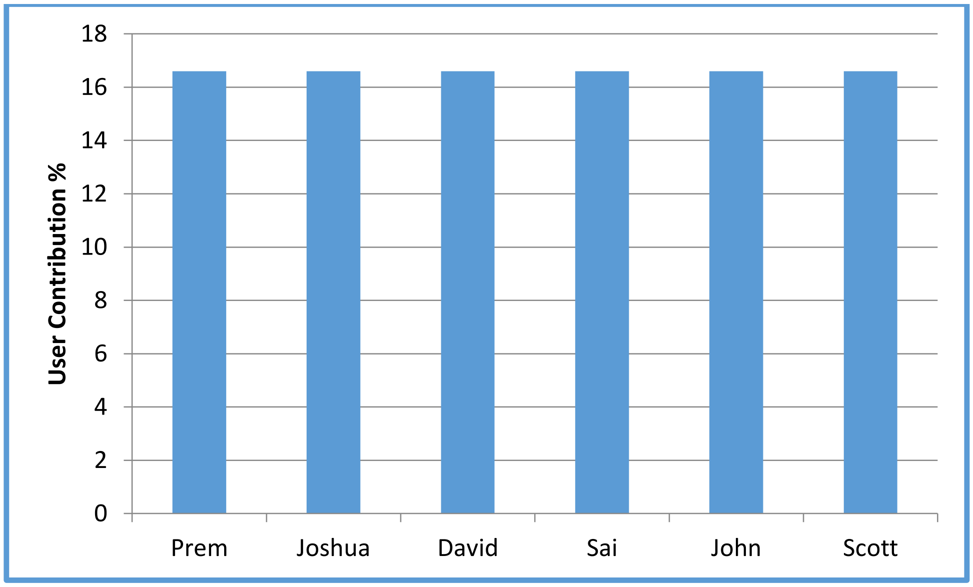


| ID | Task Name | Start | Finish | Duration |
|---|---|---|---|---|
| 1 | Use Cases | 4/2/2013 | 4/3/2013 | 2d |
| 2 | Use Case Diagrams | 4/5/2013 | 4/5/2013 | 1d |
| 3 | Requirements | 4/3/2013 | 4/7/2013 | 5d |
| 4 | Theoretical Model | 4/2/2013 | 4/7/2013 | 6d |
| 5 | System Sequence Diagrams | 4/1/2013 | 4/5/2013 | 5d |
| 6 | Interaction Diagrams | 4/7/2013 | 4/13/2013 | 1w |
| 7 | Project Management | 4/5/2013 | 4/7/2013 | 3d |
| 8 | Class Diagrams and Interface Specification | 4/7/2013 | 4/13/2013 | 1w |
| 9 | Class Diagram | 4/7/2013 | 4/10/2013 | 4d |
| 10 | Data Types and Operations Signatures | 4/8/2013 | 4/11/2013 | 4d |
| 11 | Traceability Matrix | 4/9/2013 | 4/12/2013 | 4d |
| 12 | System Architecture and System Design | 4/7/2013 | 4/14/2013 | 1w 1d |
| 13 | Architectural Styles | 4/7/2013 | 4/8/2013 | 2d |
| 14 | Identifying Subsystems | 4/8/2013 | 4/10/2013 | 3d |
| 15 | Mapping Systems to Hardware | 4/9/2013 | 4/12/2013 | 4d |
| 16 | Persistent Data Storage | 4/9/2013 | 4/10/2013 | 2d |
| 17 | Network Protocol | 4/10/2013 | 4/11/2013 | 2d |
| 18 | Global Control Flow | 4/10/2013 | 4/12/2013 | 3d |
| 19 | Hardware Requirements | 4/11/2013 | 4/11/2013 | 1d |
| 20 | Project Management | 4/13/2013 | 4/14/2013 | 2d |
| 21 | Full Report 2 | 4/20/2013 | 5/5/2013 | 2w 2d |
| 22 | Algorithms and Data Structures | 4/24/2013 | 5/5/2013 | 1w 5d |
| 23 | User Interface Design and Implementation | 4/24/2013 | 5/5/2013 | 1w 5d |
| 24 | Design of Tests | 4/24/2013 | 5/5/2013 | 1w 5d |
| 25 | Project Management | 4/24/2013 | 5/5/2013 | 1w 5d |
| 26 | Demo #1 | 4/17/2013 | 4/19/2013 | 3d |
| 27 | Full Report 3 | 4/24/2013 | 5/5/2013 | 1w 5d |
| 28 | Demo #2 | 5/5/2013 | 5/10/2013 | 6d |
| 29 | Electronic Project Archive | 5/5/2013 | 5/10/2013 | 6d |

Due to shortage of time, most of our items are being split among group members and worked on in parallel.  Once we finish the documentation for reports two and three, we plan on using the rest of semester to integrate our GUIs for the final demo.

## 12.4 Breakdown of Responsibilities

| Task | Prem | David | John | Joshua | Scott | Sai |
|---|---|---|---|---|---|---|
| Statement of Requirement | X | X | X | X | X | X |
| System Requirements | X | X | X | X | X | X |
| Mock Up Interfaces | X | X | X | X | X | X |
| Functional Requirements | X | X | X | X | X | X |
| Use Cases | X | X | X | X | X | X |
| Fully Dressed Descriptions | X | X | X | X | X | X |
| Domain Analysis | X | X | X | X | X | X |
| Mathematical Model | X | X | X | X | X | X |
| Interaction Diagrams | X | X | X | X | X | X |
| Class Diagrams and Interface | X | X | X | X | X | X |
| System Architecture & Design | X | X | X | X | X | X |
| Algorithms | X | X | X | X | X | X |
| Data Structures | X | X | X | X | X | X |
| UI Design & Implementations | X | X | X | X | X | X |
| Design of Tests | X | X | X | X | X | X |
| Project Management | X | X | X | X | X | X |
| History Of Work | X | X | X | X | X | X |
| References | X | X | X | X | X | X |



The above chart summarizes the contributions from various team members in terms of effort. The course website states to quantify the breakdown to what person did what for each section. However, we the group, would like to state that all members have equally contributed to the report since we believe that the website does not accurately describe contributions.

## 12.5 History of Work, Current Status, and Future Work

### 12.5.1 History Of Work

In this section of the report, we will discuss the history of our work.  This part serves as a journal of our accomplishments and highlights each milestone in our project.

**January 22nd - February 5th**
At the start of the semester, we initially worked on the personal health monitoring project.  We began by looking at health monitoring devices and thinking of ways to integrate these devices to create a system that would promote healthy living to the general public.  The three aspects of personal health we decided to focus on were diet, exercise, and sleep.  Then, we chose to purchase the Bodymedia Fit as well as the Motorola MotoActv.  These devices would be used to monitor the wearer's calories burned, exercise periods, and sleep patterns.  We decided to design a system which would integrate these three aspects of personal health, and interact with users to encourage them to improve in these three areas of their lives.  On February 5th, we submitted our proposal.

**February 6th - February 23rd**
During this time, our group focused on refining our ideas and writing the first report.  We received approval of our proposal from the professor and proceeded to formulate use cases and functional requirements.  We also created system sequence diagrams.  On February 23rd, we submitted our first report.

**February 24th - March 12th**
During this time, we began planning for the arrival of our devices and designing our system.  We created a webpage for our system's web application.  Afterwards, while waiting for our devices to arrive, we began working on the second report.  We received the health monitoring devices on March 12th.

**March 13th - March 29th**
On March 13th, we received an email from the professor stating that he was unsatisfied with our project ideas after reviewing our first report.  After a lengthy group meeting, we revamped our project with some new ideas, which we emailed to the professor for review.  These included: a workout aid to help users track workouts and repetitions, hydration notifications, and a social workout gaming system which used GPS to record routes and find local workout buddies.  From here, we worked closely with the professor to refine our ideas and to make sure he was satisfied with our project.  Despite the professor liking our new ideas, he determined that we would not have enough time to collect sufficient data for a working demo.  We decided to scrap the project and start over with a new

topic. The professor granted us an extension for the first demo and we chose to work on restaurant automation.

**March 30th - April 6th**
We began our work on the restaurant automation project by reading reports from previous years and brainstorming new ideas to contribute to these systems. Our main idea was to implement an inventory system which will simplify the jobs of restaurant employees and improve customer service and satisfaction. Then, we formally drafted these ideas into a theoretical model, which included the reasoning behind our ideas, algorithms/pseudocode, and specific user scenarios. On April 6th, we submitted this theoretical model to the professor for feedback, before continuing with the official first report.

**April 7th - April 14th**
During this time, our group focused on finishing the first report. After receiving positive feedback on our theoretical model, we refined our ideas and description of our system, as well as incorporated UML diagrams and use cases to help illustrate the concepts behind our product. We drafted our Customer Statement of Requirements, Glossary of Terms, Functional Requirements, Effort Estimation, and Domain Analysis. These sections were divided amongst individuals in our group to work on and later assembled into our report. On April 14th, we submitted a draft of our first report.

**April 15th - April 16th**
The professor responded quickly and provided helpful feedback on our draft of the first report. On April 16th, after making a number of changes to our draft, we submitted our final version of the first report for grading.

**April 17th - April 19th**
During this time, our group furiously programmed to put together a demo. We divided up the project into 5 individual parts: the chef’s GUI, the customer’s GUI, the waiter’s GUI, the menu item rating GUI, and the inventory system backend. The customer’s GUI displays the menu items currently available at the restaurant and allows the customer to select these items and place his or her order. The order is then submitted and queued so that it can be made by the restaurant chefs. The chef’s GUI enables customer orders to be viewed by the chef, who then selects which orders to cook. A notification is given when the order is ready to be served. The waiter’s GUI enabled waiters to view a list of orders which are finished cooking and ready to be served. Based on the order, the GUI then informs the waiter which table the order belongs to. The rating GUI allows customers to rate menu items that they had ordered. The GUI also allows both managers and customers to view past ratings to determine the popularity of specific menu items. The ratings are stored in our database. Our inventory system keeps track of ingredients stored in the restaurant’s inventory. Notifications for restocking are sent to the manager when a certain item’s stock is below a predetermined threshold. The system automatically updates item quantities when a restock is issued or when ingredients expire. The inventory system also keeps track of ingredient usage rates and sets new restocking thresholds based on this data. Due to the rushed nature of this demo, many parts of the program were hard-coded and the individual components did not interact with one another. On April 19th, we presented our demo to the professor and discussed future work for the final demo.

**April 20th - April 23rd**
During this time, we cleaned up our project code and assembled our work into one unit for our demo submission. On April 23rd, after documenting the code and placing instructions and examples in various readme files, we uploaded our Demo 1 submission to Sakai. During this time, we also began work on our second report.

**April 24th - May 5th**
During this time, we continued working on the second report. With the end of semester rapidly approaching, we decided to lump everything together and work on the third report as well. Our goal was to get all the documentation out of the way so that we would have the remainder of the semester to work on coding for the final demo. For these reports, we worked on Interaction Diagrams, Class Diagrams, Interface Specification, System Architecture, System Design, Algorithms and Data Structures, User Interface Design/Implementation, Design of Tests, and our References. We divided these sections for individual group members to complete, and compiled everything into our second report for submission. Then, we merged the first and second reports and reviewed everything to create the third report. On May 5th, we submitted both our second report and our final (third) report.

### 12.5.2 Current Status

In this section, we will discuss the current status and features of our project after the first demo and our submission of the three reports.

Our system hasn't changed much since the first demo. We have implemented the following features:

- The system keeps track of ingredient inventory and notifies the manager of when an ingredient restock is needed.
- The system provides the customer with a menu GUI, which displays a list of the menu items that are currently able to be made at the restaurant, based on ingredient availability. The customer interacts with this GUI to view ratings on these menu items, as well as place their order. Afterwards, the customer can submit his or her own rating of the menu item(s) ordered.
- The system provides the chef with a GUI which displays a queue of orders submitted by restaurant customers. The chef can select which order(s) he or she wishes to cook from this queue, and the GUI will display the recipe and ingredients needed to make the selected item(s). Once the chef is finished making the item(s), a notification will be sent to the waiter to serve the dish, and the order will be queued for delivery. The restaurant inventory will also be reduced by the appropriate amount, based on which ingredients were used.
- The system provides the waiter with a GUI which displays a queue of customer orders that are cooked and ready to be delivered. The waiter can select which order(s) to deliver from this queue, and the GUI will display the table to which the order belongs. Then the order will be dequeued from this list and the customer will be allowed to rate the item on the customer's GUI.

### 12.5.3 Future Work

In this section, we will discuss our goals for the final demo, as well as future ideas that can extend and improve our current system.

For the upcoming final demo, our main priority is to combine all of the individual components we implemented in our first demo to have a single, integrated system. Some challenges we face in doing so are:

- **Different programming languages.** Because we were short on time for the first demo, we split the work and completed each part individually. Each of us chose different programming languages and different methods of implementation. Integrating these components will be a challenge since they were developed separately. Some parts may even need to be completely redone to make every piece function together as a cohesive unit.
- **Communication between modules.** Our system involves many different different modules which have different responsibilities, but also communicate with each other extensively. Each module depends on the actions of at least one other module, and thus we must take care to ensure proper communication between these modules when integrating our project.

For future work, there are some things that we thought of which would further automate restaurant service and/or improve the current system:

- **Waiter notifications.** Even during relatively calm business hours, waiters are not completely available to service customers' needs immediately. Usually, customers must wait for the waiter/waitress to walk by their table before attempting to get their attention. To facilitate this process, we can implement a notification system that will allow customers to push a button when they want a waiter to come to their table. This will be similar to an airline's flight attendant notification button, and will eliminate the need for customers to constantly keep an eye out for a waiter when they require the waiter's services.
- **Automatic delivery.** Instead of having a waiter to deliver cooked orders, we thought of having a automatic delivery system. This is an improvement because during busy hours, waiters are often have many tables to attend to, and customers end up having to wait longer to be served despite their food being ready. Having an automatic delivery system will free up waiters to tend to other customer needs, improving customer satisfaction and reducing the workload of the waiter. To implement this, we could integrate our project with the Robotic Delivery System project from Fall 2012.
- **Video reviews.** Many online products feature video reviews, which demonstrate hands-on interaction between the consumer and the product. In order to further engage our customers and provide better reviews on our menu items in the future, we could add an optional video reviewing portion to our customer GUI.

# References

[1] http://www.enewsbuilder.net/peoplereport/e_article000657699.cfm?x=b11,0,w (Robot Picture)

[2] http://www.ece.rutgers.edu/~marsic/books/SE/projects/Restaurant/2012-g11-report3.pdf (Group 2, 2011 report)

[3] http://en.wikipedia.org/wiki/Event-driven_architecture

[4] http://msdn.microsoft.com/en-us/library/ee658117.aspx

[5] http://codethat.files.wordpress.com/2010/01/scheme.png