Software Engineering
Group #9
Restaurant automation
https://github.com/kevinlin47/Software-Engineering-Project

2/26/17
Katie Bruett
Anthony Caivano
Rahul Gupta
Patrick Karol
Kevin Lin
Kimberly Ngai

	Team Member Name					
	Katie Bruett	Anthony Caivano	Rahul Gupta	Patrick Karol	Kevin Lin	Kimberly Ngai
Project management (18 Points)		3%	97%			
Sec.1: Interaction Diagrams (30 points)					99%	1%
Sec.2: Classes + Specs (10 points)		90%				10%
Sec.3: Sys Arch & Design (15 points)	70%	1%			19%	10%
Sec.4: Alg's & data struct (4 points)		100%				
Sec.5: User Interface (11 points)	100%					
Sec.6:Testing design (12 points)				50%	40%	10%

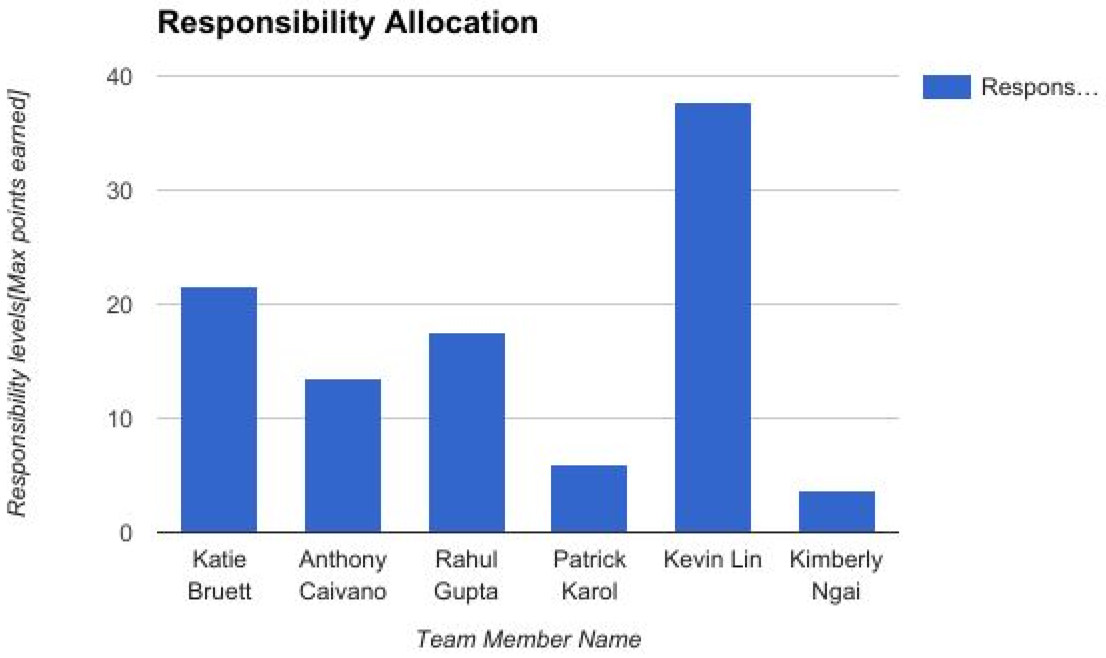
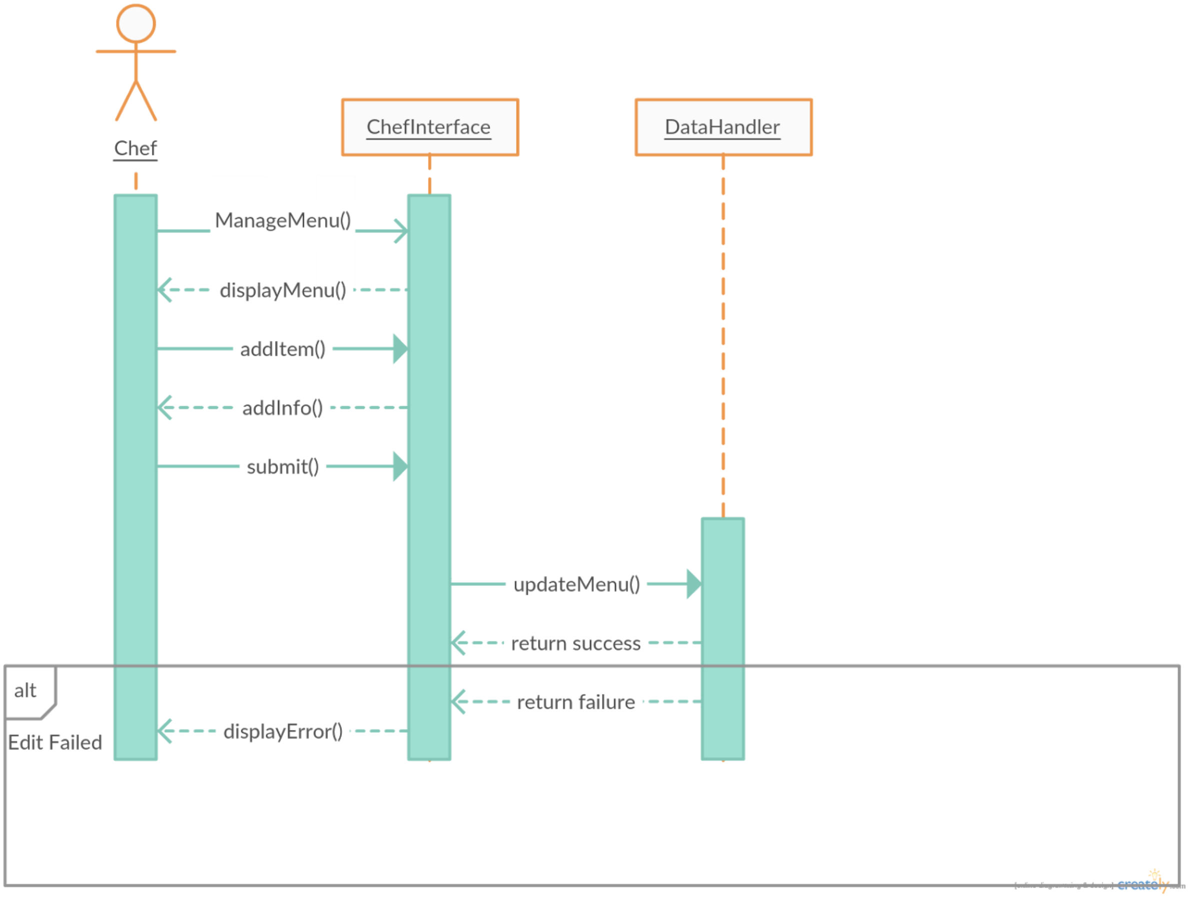

Responsibility Allocation
Responsibility levels[Max points earned]
40
30
20
10
0
Katie Bruett
Anthony Caivano
Rahul Gupta
Patrick Karol
Kevin Lin
Kimberly Ngai
Team Member Name
Respons…

Table of Contents

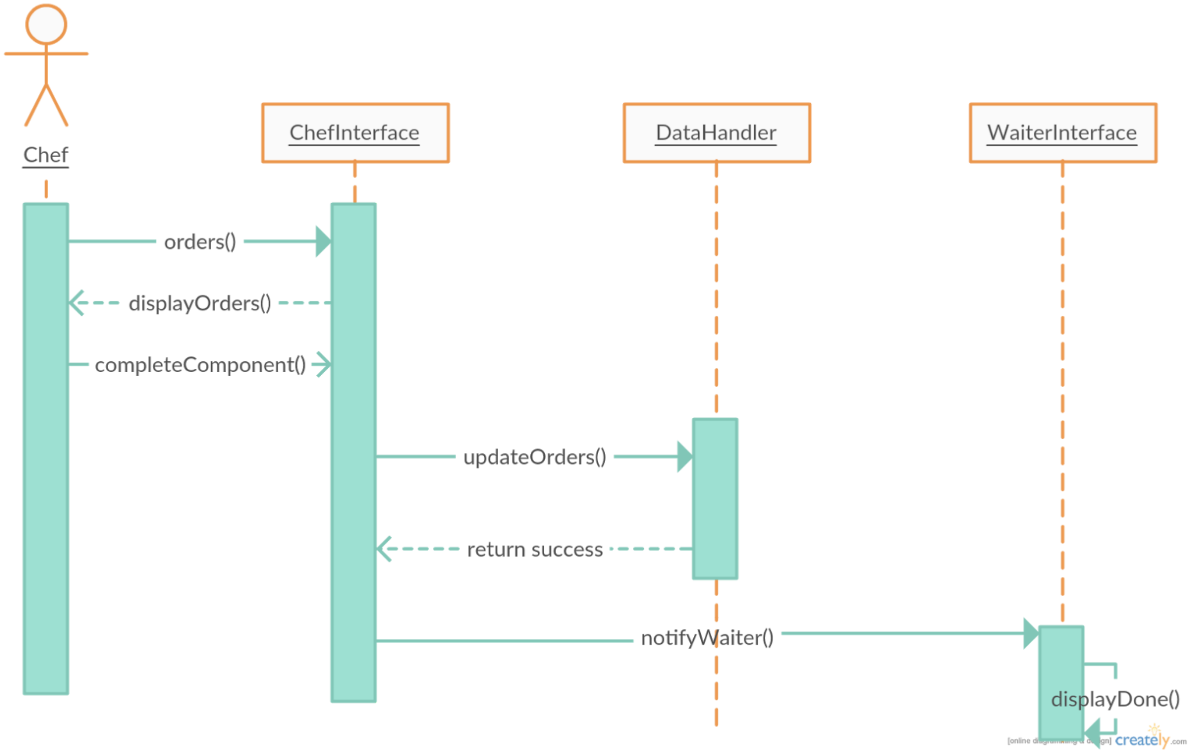
1. Interaction Diagrams

1.1 UC-6 (Edit Menu)



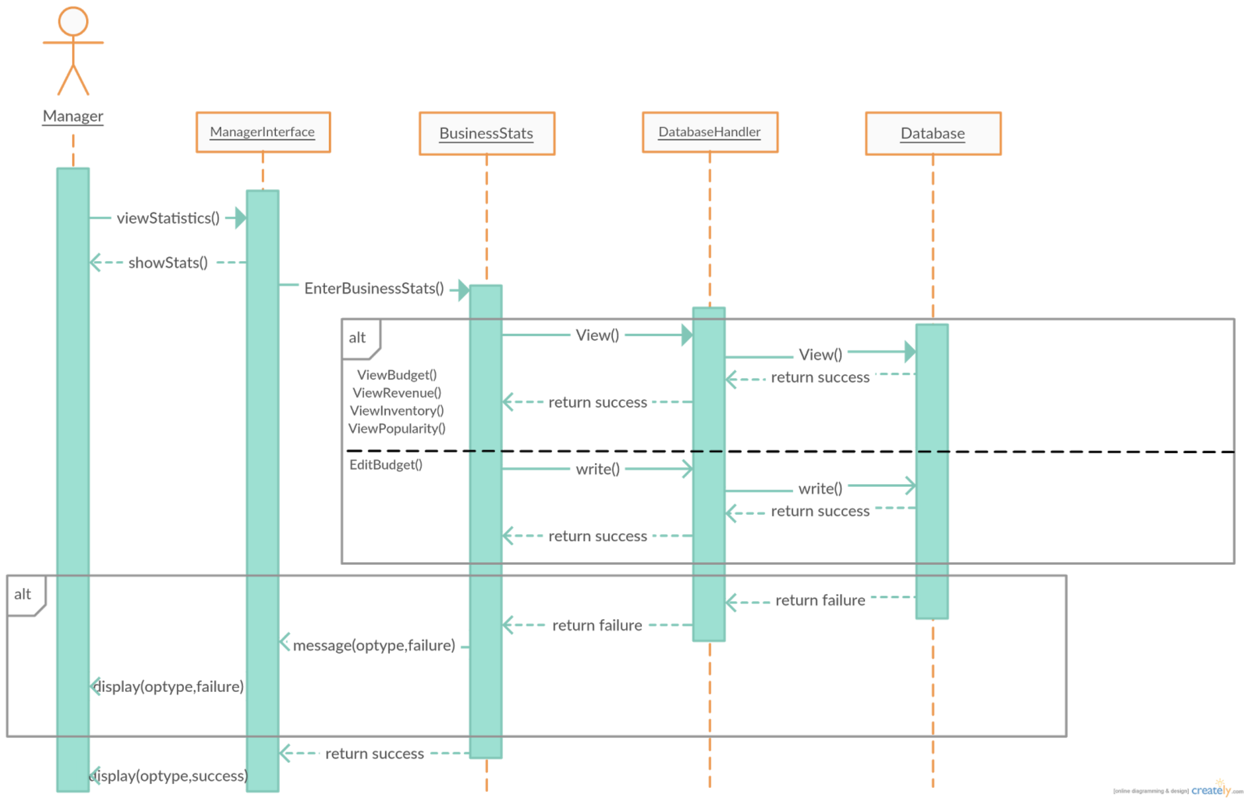
The chef can log into his UI, and then select the manage menu option. From there he will be shown a list of all the current menu items. He can add items to the menu by entering required information about the new item, such as name, ingredients, and price. Existing menu items can also be updated in terms of price or ingredients. The chef then clicks submit and the menu database is updated if everything is handled correctly, but will display an error message if the menu did not update correctly.

1.2 UC-9 (Order Done)



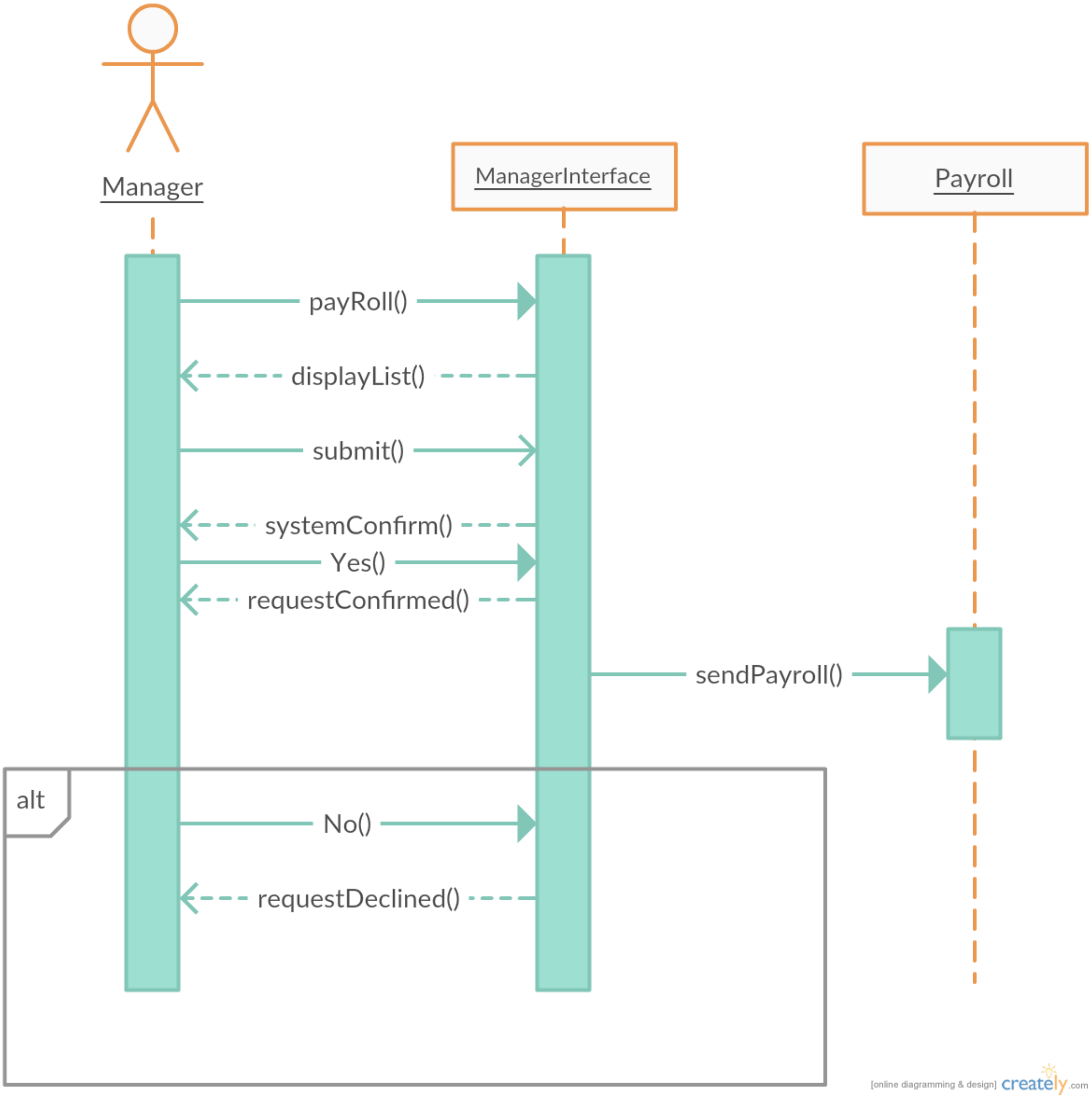
Once logged into his UI, the chef can access the orders queue. From there he will be shown the queue of current orders. Once he is done with a specific order, he can notify the waiter that an order is complete. The waiter's UI will then display that an order is done. The order queue updates to show the remaining orders left.

1.3 UC-19 (View Statistics)



The manager is logged into his UI and he chooses the view statistics option. He is then shown a list of statistics he can choose to look at (budget, revenue, inventory, popularity). He selects one of the choices, and is shown the requested information pulled from the database, with a success message. If for some reason the request fails, due to possibly not pulling information from the database, the manager will be shown a failure message. Alternatively, the manager can also edit the budget when accessing the budget statistic.

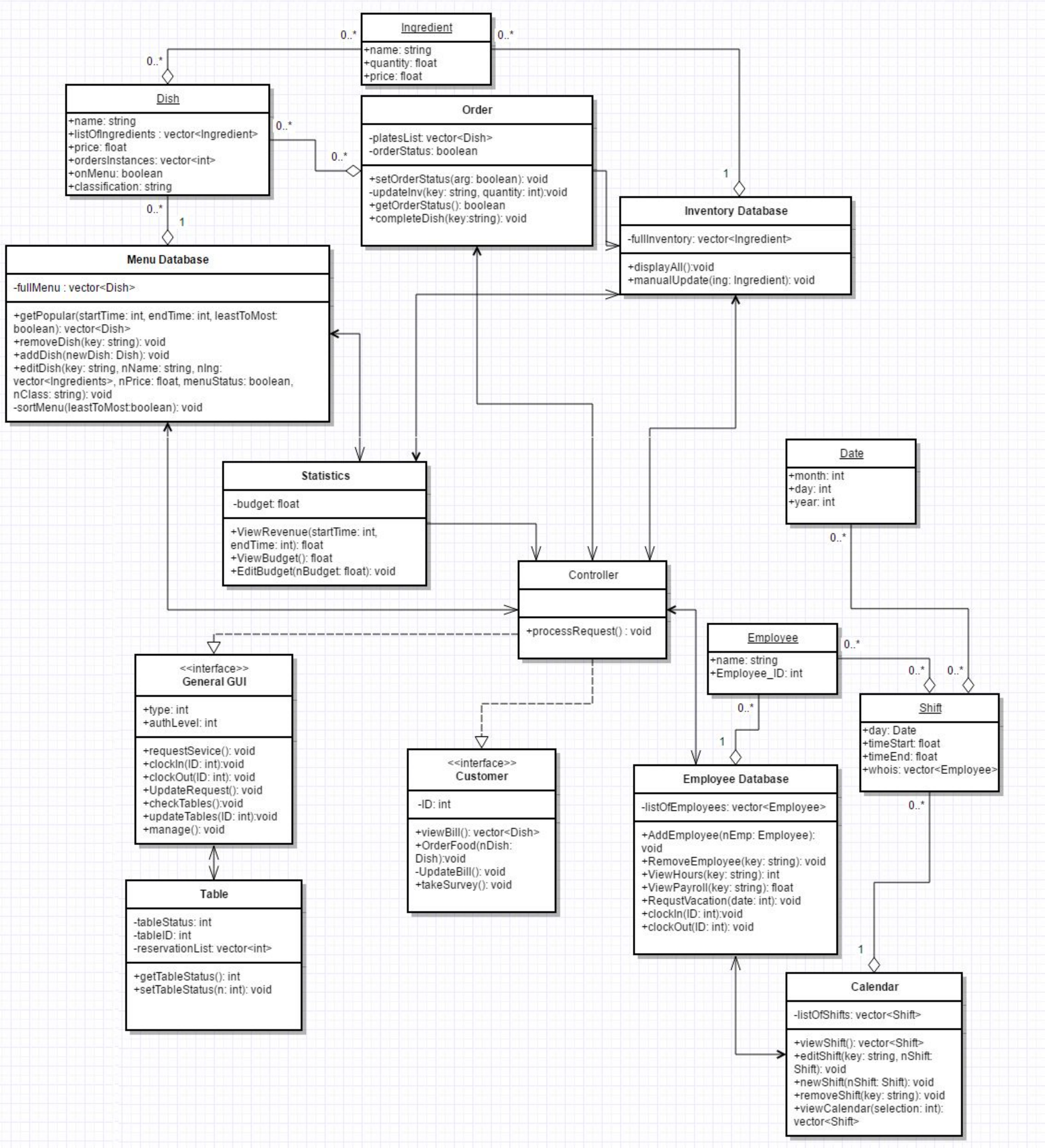
1.4 UC-22 (Payroll Status)



The manager is logged into his UI, he chooses the employee option, and is brought to the payroll page. The list of employees is shown with their current hours worked, their hourly rate, and their pay for the week. The manager submits the payroll which prompts the system to ask for confirmation. If the manager hits yes then the payroll is sent to whoever handles and distributes the payroll. If the manager hits no the payroll does not get sent.

2. Class Diagram and Interface Specification

2.1 Class Diagram


Ingredient
+name: string
+quantity: float
+price: float
0..*
0..*
0..*
Dish
+name: string
+listOfIngredients : vector<Ingredient>
+price: float
+ordersInstances: vector<int>
+onMenu: boolean
+classification: string
0..*
0..*
Order
-platesList: vector<Dish>
-orderStatus: boolean
+setOrderStatus(arg: boolean): void
-updateInv(key: string, quantity: int):void
+getOrderStatus(): boolean
+completeDish(key:string): void
1
Inventory Database
-fullInventory: vector<Ingredient>
+displayAll():void
+manualUpdate(ing: Ingredient): void
0..*
1
Menu Database
-fullMenu : vector<Dish>
+getPopular(startTime: int, endTime: int, leastToMost: boolean): vector<Dish>
+removeDish(key: string): void
+addDish(newDish: Dish): void
+editDish(key: string, nName: string, nIng: vector<Ingredients>, nPrice: float, menuStatus: boolean, nClass: string): void
-sortMenu(leastToMost:boolean): void
Statistics
-budget: float
+ViewRevenue(startTime: int, endTime: int): float
+ViewBudget(): float
+EditBudget(nBudget: float): void
Date
+month: int
+day: int
+year: int
0..*
Controller
+processRequest() : void
Employee
+name: string
+Employee_ID: int
0..*
0..*
0..*
0..*
1
Shift
+day: Date
+timeStart: float
+timeEnd: float
+whois: vector<Employee>
<<interface>>
General GUI
+type: int
+authLevel: int
+requestSevice(): void
+clockIn(ID: int):void
+clockOut(ID: int): void
+UpdateRequest(): void
+checkTables():void
+updateTables(ID: int):void
+manage(): void
<<interface>>
Customer
-ID: int
+viewBill(): vector<Dish>
+OrderFood(nDish: Dish):void
-UpdateBill(): void
+takeSurvey(): void
Employee Database
-listOfEmployees: vector<Employee>
+AddEmployee(nEmp: Employee): void
+RemoveEmployee(key: string): void
+ViewHours(key: string): int
+ViewPayroll(key: string): float
+RequstVacation(date: int): void
+clockIn(ID: int):void
+clockOut(ID: int): void
0..*
1
Table
-tableStatus: int
-tableID: int
-reservationList: vector<int>
+getTableStatus(): int
+setTableStatus(n: int): void
Calendar
-listOfShifts: vector<Shift>
+viewShift(): vector<Shift>
+editShift(key: string, nShift: Shift): void
+newShift(nShift: Shift): void
+removeShift(key: string): void
+viewCalendar(selection: int): vector<Shift>

2.2 Data Types and Operation Signatures

2.2.1 Dish Object Type

The Dish object is the baseline object for what a customer will order, as well as what is stored in the menu.

Attributes:

+name: string - A simple string that has the name of the dish

+listOfIngredients : vector<Ingredient> - A vector of Ingredients that are in the dish

+price: float - The price, in dollars of the dish.

+ordersInstances: vector<int> - A vector of UNIX times of each instance when a dish was ordered.

+onMenu:boolean - Indicates if the dish can be seen on the menu.

+classification: string - Indicates what the meal is - for example: breakfast, appetizer, etc.

2.2.2 Ingredient Object Type

The Ingredient object is the baseline object for what a dish is comprised of, and what is kept track of by the inventory database.

Attributes:

+name: string - A simple string that has the name of the ingredient.

+quantity: float - The amount that the restaurant has in stock

+price: float - The cost of the ingredient

2.2.3 Menu Database

This class keeps track of each dish on the menu, and also is able to determine the most popular item on it.

Attributes:

-fullMenu: vector<Dish> - Array of every item on the menu.

Methods:

+getPopular(startTime: int, endTime: int, leastToMost: boolean): vector<Dish> - Used to determine which dishes are the most popular within a given timeframe.

+removeDish(key: string): void - Allows the user to remove a dish from the menu.

+addDish(newDish: Dish): void - Allows the user to add a dish to the menu.

+editDish(key: string, nName: string, nIng: vector<Ingredients>, nPrice: float, menuStatus: boolean, nClass: string): void - Allows the user to edit a dish already on the menu.

-sortMenu(leastToMost:boolean): void - called by getPopular() to sort by popularity.

2.2.4 Order

This class allows the user to place an order, and the chef to mark it as complete when finished.

Attributes:
-platesList: vector<Dish> - Vector of the dishes in the order.
-orderStatus: boolean - Indicates if the order is complete or not
Methods:
+setOrderStatus(arg: boolean): void - Allows the chef to set the order as complete
-updateInv(key: string, quantity: int):void - Updates the inventory when the order is completed
+getOrderStatus(): boolean - Returns orderStatus.
+completeDish(key:string): void - Allows the chef to complete a dish.

2.2.5 Inventory Database

This database keeps track of the restaurant's inventory, which can update automatically or manually.
Attributes:
-fullInventory: vector<Ingredient> - A full list of the whole inventory
Methods:
+displayAll(): inv:vector<Inventory> - Returns fullInventory
+manualUpdate(ing:Ingredient): void - Allows the user to update the inventory manually, subverting Order's updateInv().

2.2.6 Statistics

This class can access the menu and inventory databases for easy access to menu statistics, along with budget information.
Attributes:
-budget:float - The dollar amount, per week, that the restaurant can spend.
Methods:
+ViewRevenue(startTime: int, endTime: int): float - Calculates the revenue for a specified period of time.
+ViewBudget(): float - Returns the total budget remaining for the week.
+EditBudget(nBudget: float): void - Allows the user to manually update the budget.

2.2.7 Controller

Currently, this is a generic class that will eventually process requests between classes.
Methods:
+processRequest():void - Functionally not roughly defined, but allows for communication between classes.

2.2.8 Date Object Type

This is a data structure that allows the calendar and shift to simply access the data without dealing with UNIX time.
Attributes:
+month: int - The month, {1-12}.
+day: int - The day of the month, {1-31}.

+year: int - The current year.

2.2.9 Employee Object Type

This object contains employee information
Attributes
+name: string - The name of the employee
+Employee_ID: int- An identification number for the employee.
(not shown) +hoursWorked: int - The number of hours the employee worked in the past week.
(not shown) +tempClockInTime: int - The clock in time for the day.
(not shown) +tempClockOutTime: int - The clock out time for the day.

2.2.10 Shift Object Type

This object contains the date, time, and employees assigned to the shift.
Attributes
+day: Date - The date the shift is taking place.
+timeStart: float - The start time of the shift.
+timeEnd: float - The end time of the shift.
+whois: vector<Employee> - List of employees who are assigned to the shift.

2.2.11 Calendar

This class allows users to view several shifts simultaneously, as well as edit and add new ones.
Attributes
-listOfShifts: vector<Shift> - A list of every shift on the calendar.
Methods
+viewShift(): vector<Shift> - Allows the user to view a shift.
+editShift(key: string, nShift: Shift): void - Allows the user to edit a shift.
+newShift(nShift: Shift): void - Allows the user to add a new shift.
+removeShift(key: string): void - Allows the user to search and delete a shift, if it exists.
+viewCalendar(selection: int): vector<Shift> - Displays the whole calendar.

2.2.12 Employee Database

This class holds information about every employee, the total payroll, and other information.
Attributes
-listOfEmployees: vector<Employee> - A list of every employee.
Methods
+AddEmployee(nEmp: Employee): void - Adds a new employee to the existing list.
+RemoveEmployee(key: string): void - Searches for and removes an employee from the list, if found.
+ViewHours(key: string): int - Returns the amount of hours an employee has worked, if found.

+ViewPayroll(key: string): float - Returns the payroll data.
+RequstVacation(date: int): void - Allows the user to make a vacation request.
+clockIn(ID: int):void - Allows the user to clock in for the day.
+clockOut(ID: int): void - Allows the user to clock out for the day.

2.2.13 Table

This class has information regarding each table in the restaurant.

Attributes
-tableStatus: int - Enumerated variable that indicates if the table is vacant, occupied, dirty, or out of service.
-tableID: int - Identification for a table.
-reservationList: vector<int> - List of UNIX times where the table will be reserved by someone.

Methods
+getTableStatus(): int - Returns tableStatus.
+setTableStatus(n: int): void - Sets tableStatus.

2.2.14 General GUI Interface

This interface will serve as the general GUI on the main screen of the application, and have different levels of access depending on authentication level. This is currently not all-encompassing.

Attributes
+type: int - Indicates if the user is a customer, employee, etc.
+authLevel: int - Gives a more concrete specification of type.

Methods
+requestSevice(): void - Allows an employee to request service for a particular object in the restaurant.
+clockIn(ID: int):void - Allows the employee to clock in.
+clockOut(ID: int): void- Allows the employee to clock out.
+UpdateRequest(): void - Allows a manager to update a request for service or vacation.
+checkTables():void - Informs the user of the tables in the restaurant.
+updateTables(ID: int):void - Allows the user to update table information, based on its ID number.
+manage(): void - General management function, ill defined.

2.2.15 Customer Interface

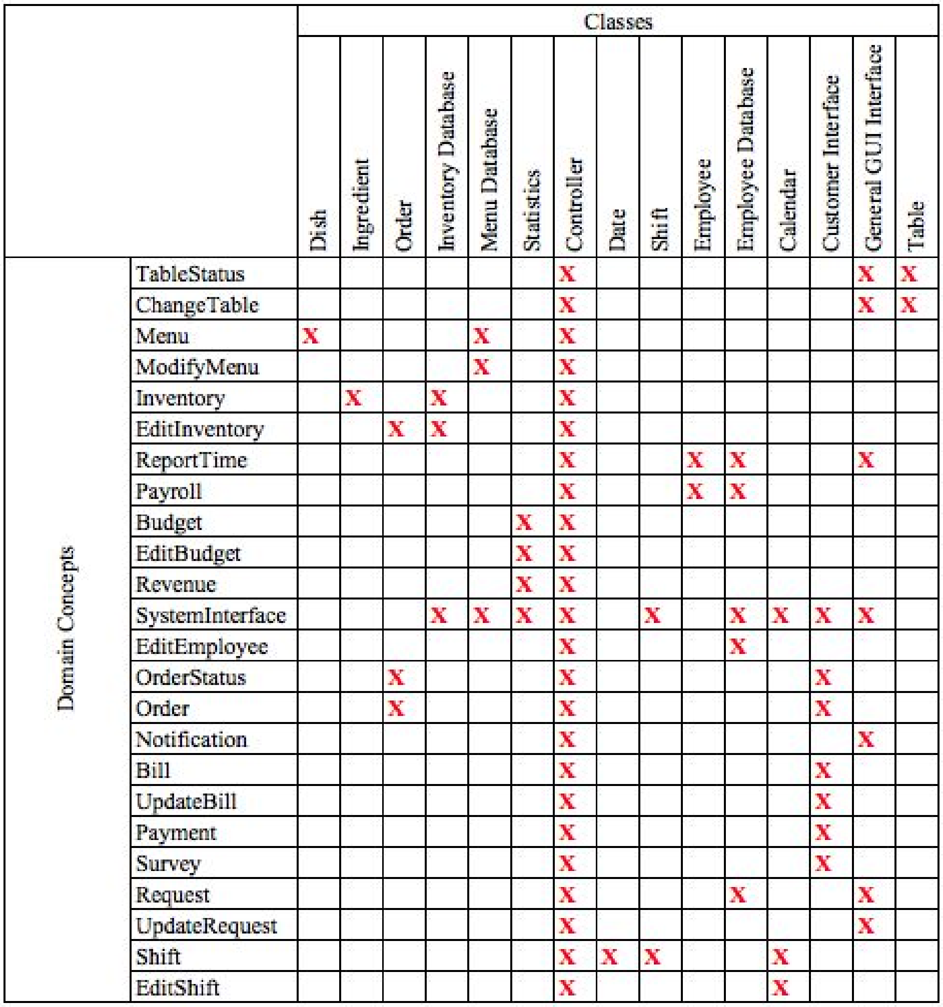
This is a more specific version of the General GUI, that provides streamlined information for the customers.

Attributes
-ID:int - Customer identification number.

Methods
+viewBill(): vector<Dish> - Allows the user to check the status of his order.
+OrderFood(nDish: Dish):void - Allows the user to send an order request to the kitchen.

-UpdateBill(): void - Called by OrderFood() to update the bill after an order.
+takeSurvey(): void - Allows the user to take a survey of the quality, etc. of the restaurant.

2.3 Traceability Matrix



		Classes														
		Dish	Ingredient	Order	Inventory Database	Menu Database	Statistics	Controller	Date	Shift	Employee	Employee Database	Calendar	Customer Interface	General GUI Interface	Table
Domain Concepts	TableStatus							X							X	X
	ChangeTable							X							X	X
	Menu	X				X		X								
	ModifyMenu					X		X								
	Inventory		X		X			X								
	EditInventory			X	X			X								
	ReportTime							X			X	X			X	
	Payroll							X			X	X				
	Budget						X	X								
	EditBudget						X	X								
	Revenue						X	X								
	SystemInterface				X	X	X	X		X		X	X	X	X	
	EditEmployee							X				X				
	OrderStatus			X				X						X		
	Order			X				X						X		
	Notification							X							X	
	Bill							X						X		
	UpdateBill							X						X		
	Payment							X						X		
	Survey							X						X		
	Request							X				X			X	
	UpdateRequest							X							X	
	Shift							X	X	X			X			
	EditShift							X					X			

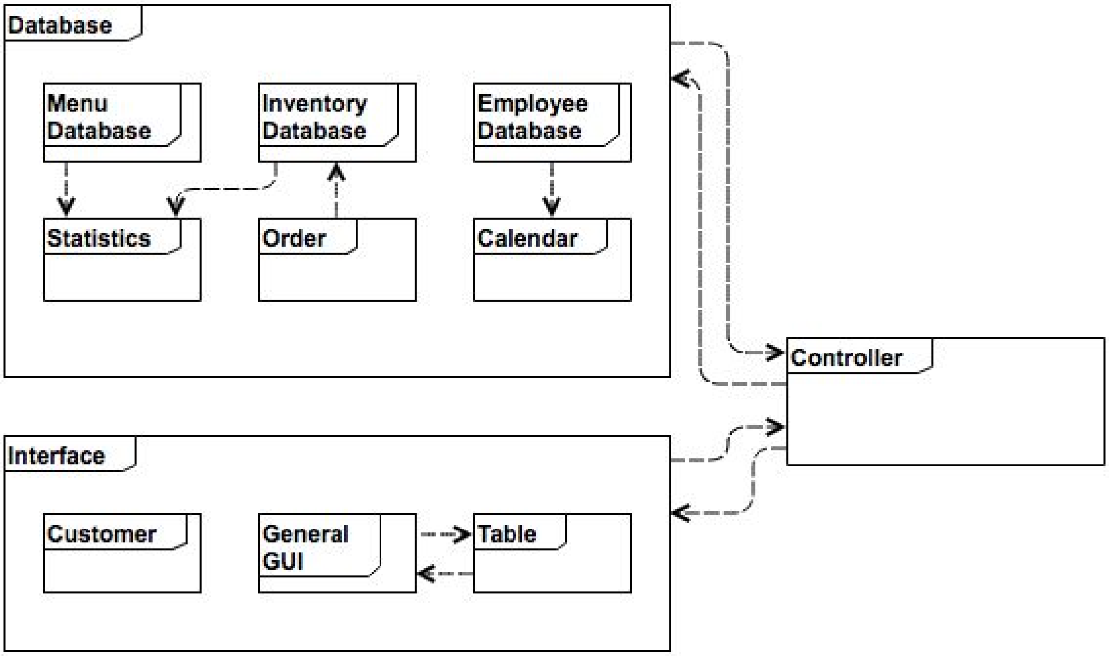
A majority of the domain concepts were lumped together into separate classes to further show which users would use which domain, e.g. a generalized Menu Database was derived from grouping all the menu concepts together. Since the Controller handles all requests and changes, it was derived from all the domain concepts. The remaining classes were derived from the concepts using common sense and generalizations.

3. System Architecture and System Design

3.1 Architectural Styles

Our project utilizes a number of architectural styles in its design. The two main styles that we implemented were the Client/Server style and the Object-Oriented Style. The Client/Server style is basically a separation of the system into two interconnecting parts. In our system, it will feature a desktop user interface which would communicate to a database containing all the relevant information about the restaurant's kitchen, management, and customer service. This is also known as a 2-Tier architecture style. The system will grant access to certain data based on user identification, therefore increasing the security of the system. This system also boasts a simplicity factor based on the fact that all the relevant data is stored on one central server. The other main style we utilized is the Object-Oriented style. This method is all about individual objects being independent from other objects, making the system more comprehensive and easy to understand. In our system, these independent objects will be all the different options on the main menu of the system. By separating the kitchen tasks from the managerial, it improves simplicity and efficiency for both the user and the system in general. These styles work together to make up our system architectural style.

3.2 Identifying Subsystems



Based on the class diagram, the system will have three packages: Interface, Database, and Controller. The Interface package is comprised of the Customer and the General GUI Interfaces where both Interfaces can access/input different options based on the user. The Database contains all the information needed to run the restaurant, e.g. the menu and employee information, and also information regarding dishes, ingredients, and any business statistics. The last package is the Controller which acts as the mediator between the Database and the Interface. The controller is in charge of handling all requests made from the interfaces and transfers any information between the Database and Interface.

3.3 Mapping Subsystems to Hardware

Our system will need to be run on multiple computers as the system is based around the idea of each table of customers having their own system, the kitchen having one, the waiters having one, and the manager having his/her own. These would be identifiers of the subsystems. The main server will be used by the manager on a computer system, with granted access to the entire system. The customer subsystem would be in place on the tablet given to each table and then would be given access to the menu, bill, and survey. The waiter subsystem would be on the tablets used by the waiters and given access to the menu and bill also, as well as ability to send and track orders with the kitchen. Lastly, the kitchen subsystem would be on a computer system within the kitchen with access to orders, inventory, menu, and a communication link to the waiter subsystem to notify waiters of completed orders.

3.4 Persistent Data Storage

Data will need to be saved by this system in order to check for inventory, menu, and employee information. The data will be saved into separate plain text files, for each classification.

The inventory information will need to contain a counter for the amount of ingredients included in the file, along with a name, quantity, and price for each ingredient.

The menu data will be significantly more complicated, as it will require a counter for the amount of dishes on the menu, a string for a name, the number of ingredients in the dish, a list of ingredients in the dish (name only), the price, the number of times the dish has been ordered, a list of times of when the dish was ordered, an indicator if the dish is on the menu, and the dish's classification.

The employee data will contain a counter for each employee in the database, along with the name, identification number, and payroll information.

A sample file format for each is shown below:

Inventory:

```
      2
      Butter,2,1.50
      Onion,20,0.75
```

Menu/Dish:

```
      1
      Chicken parmesan,3,chicken breast,tomato
sauce,mozzarella,7.99,2,792893601,763690401,1,entree
```

Employee:

```
2
Bill Preston,403,<payroll data>
Theodore Logan,18,<payroll data>
```

3.5 Network Protocol

Our system will need to connect with other computers, for example waiter and chef interactions in the system. Our system will most likely use Java sockets because it seems the most efficient way of doing so without requiring extra developer time to be used on network protocol that could be used on other parts of the system. An example of messages being sent and format are as such. The waiter sends an order to the chef where it will be shown on the order queue on the chef's system. The new order will be displayed at the end of the queue, and display to the chef's screen. Once an order is complete, the chef signals in the system that a particular order is done. The queue is updated and the waiter screen will show a dialog box notifying them that the chef has completed their order.

3.6 Global Control Flow

3.6.1 Execution Orderness

Our system is a procedure-driven system. This means that the system waits for an employee or customer to go through a given set of steps in order to initiate a command. Each user, depending on their status, goes through the same steps in order to complete the task they need to accomplish. For example, if a customer wants to pay their bill, every customer will go through the same set of commands to complete their task: Click "Pay Bill", Click "Confirm", Swipe Card, Write Signature, and take the survey. This applies to each user completing each task.

3.6.2 Time Dependency

Our system works with timers in relation to the kitchen aspect as the manager and chef need to gauge how long a customer has been waiting for their order to be made and delivered to their table. So the kitchen aspect is a real-time system whereas the other sectors of the system are event-response type. Each dish is given an individual completion time estimation that the chefs utilize in order to finish all meals for one table around the same time. So since these times are varying, the system is not periodic. The constraint of time on each table is around 30 minutes from the time of ordering. The chefs will be encouraged to finish the food at a much faster pace than 30 minutes, but 30 minutes is the time of which the manager would be signaled to go to the table, apologize, give the table their meal for free, and thus cost the restaurant the price of the ingredients to make the food on the table.

3.6.3 Concurrency

Our system will utilize threads as the main goal of the system is to be able to have multiple users inputting commands and working on the system at once. If we do not utilize threads, each command would have to finish before the next would begin which would result in an inefficient restaurant. We would obviously use a different thread for manager tasks, customer tasks, and kitchen tasks as those need to be completed simultaneously. These threads would interconnect as they need to communicate commands between the three subsystems. The kitchen would also need multiple threads so that the chef could cook more than one table's food at a time.

3.7 Hardware Requirements

Our system would need to implement a color display as it will be using colors to display the cleanliness and availability of tables in the restaurant. It does not require a high resolution display as the objects on the screen will be of little detail. Since the system will be running on tablets, the screen resolution should be around 1024 x 600, or around the same as a low price tablet. The system, although relatively simple, is composed of a large database of information from the employee information to the inventory to the survey answers, so the hard drive needs to be able to contain all that information. The main computer, the one which the manager is using and stores the load of the information, should contain at least 100 GB of space in order to hold everything. The tablets, on the other hand, will wirelessly transmit their information directly to the main computer system, requiring that they only have around 2GB of hard drive space. The kitchen computer must also be able to store a large amount of information and should have around 10GB of space. Although the restaurant does not have an excessive number of employees, or tables for that matter, it will rely on a fast and efficient network bandwidth to ensure the data is transmitted quickly from one subsystem to the next, requiring a minimum of 2Mbps network bandwidth.

4. Data Structures

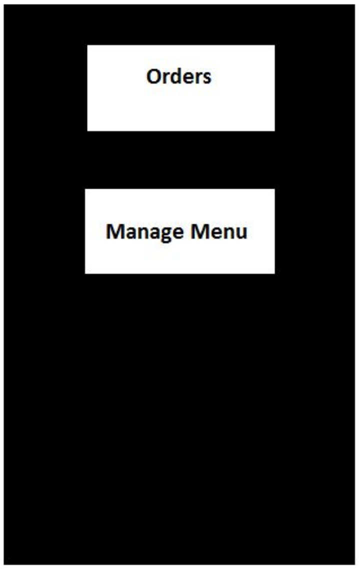
Overall, the system will use data structures that provide maximum flexibility, rather than performance. This decision was made in order to provide users with a simple and streamlined experience. From a programming standpoint, providing this experience would be easier in a flexible environment; a high-performance implementation, such as use of dynamically allocated arrays, might be awkward to use for data manipulation.

The system will store database information during runtime using std:vectors. Database management for users should allow removal, addition, and re-sorting of the items in each database, particularly for the menu and inventory databases. Furthermore, these databases will never practically reach sizes where the std:vector's performance issues come into play. As a result, vectors supply the perfect environment for database storage.

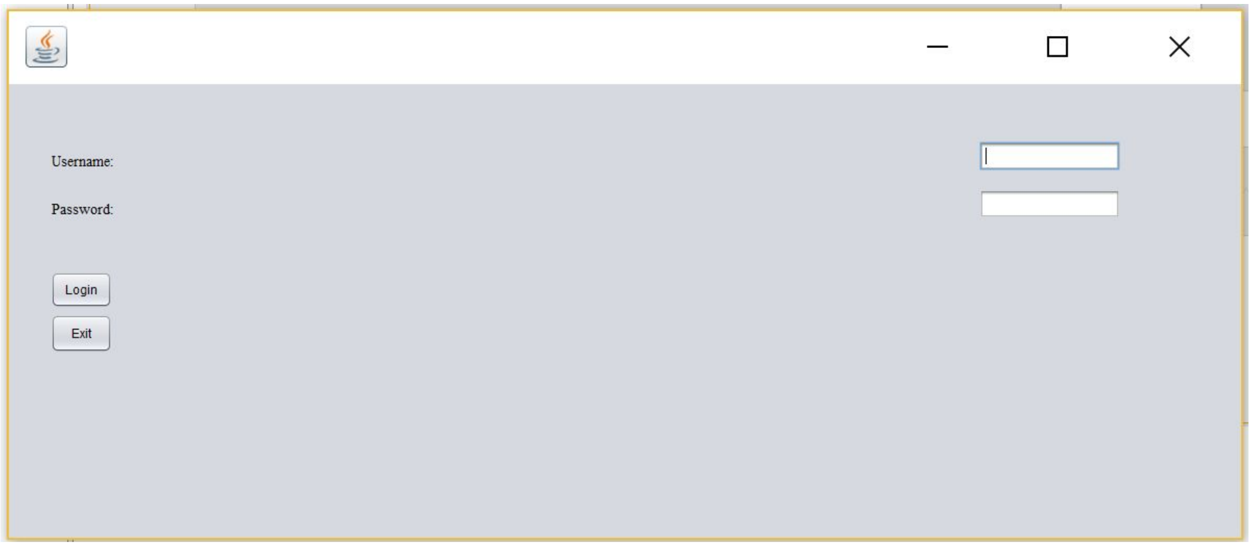
Similarly, order information will be stored in a vector as well. We want to allow chefs to tackle any dish in the "queue", so we should not limit the programming to traditional queues that follow "first in first out". The use of vectors will allow for easy deletion of an order, even if that order is in the middle of the overall queue.

5. User Interface Design and Implementation

Our implementation of the user interface design has drastically changed since report one. In our old design, it was never physically implemented, but rather an idea of what it might look like. Below is what was imagined originally for the chef to see when he/she logged into the system.



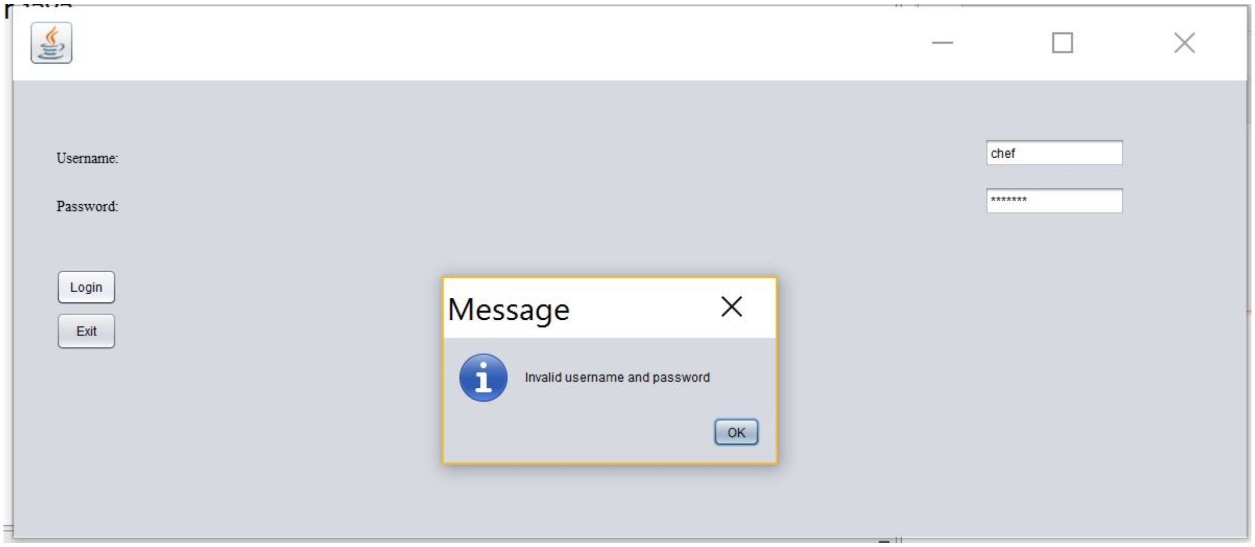
The contents described in detail in report one about what each section will contain and the general overview of how it will look will be approximately the same as before. Since then, the coding of the login screen that the users of the system will utilize has been finished, giving a clearer image of how our system will appear to the users.



This design was implemented using Java code. It is currently configured with four possible logins for entry to the system. Each login corresponds to a different type of user in the system. For now, the entries include:

User	Username	Password
Manager	manager	manager
Waiter	waiter	waiter
Chef	chef	chef
Customer	customer	customer

In actual implementation of this system, each employee would be given a simple and unique username and password combination to allow them access to whichever section of the system they are authorized to access. For now, the simple access codes for each type of user grant us a clear understanding of who is accessing the system and what they can see when they login. If a user were to enter their information incorrectly, a pop-up stating that they have entered invalid information is shown:



chef

****

Message

Invalid username and password

In this case above, the user entered "chef" for username and "manager" for password, denying them entry into the system.

As can be seen very quickly, the largest change between the old and new layout of our UI is the orientation of the screen. It was thought that a horizontal layout would allow the user to see the screen, options, and text better. This would slightly increase the ease of use of the user. In terms of the rest of the ease of use between sections of the system, the navigation of the user will be generally the same with the same options as were given in report one. Another difference that will be made since the first report will be within the menu section. The foods will be listed in categories of foods: "Drinks", "Appetizers", "Entrees", and "Desserts". These will each have a submenu for the customer, user, waiter, or chef to navigate with a clearer idea of where to find the specific type of dish they are looking for. For now, the rest of the GUI has yet to be physically implemented, but will follow the general layout and look of the above login screen, coded in Java.

6. Design of Tests

6.1 Test Cases

6.1.1 Chef

Test-Case Identifier: TC-01
Function Tested: addToMenu(string itemName, double Price): bool
Pass/Fail Criteria: Test passes if the new item is added to the menu list.

Test Procedure	**Expected Results**
Call Function(Pass)	Item will be added to the menu
Call Function(Fail)	If item already exists it will return false

Test-Case Identifier: TC-02
Function Tested: updateMenuItem(string itemName, double Price): bool
Pass/Fail Criteria: Test passes if existing item is updated.

Test Procedure	**Expected Results**
Call Function(Pass)	If item name already exists and has a different price, the menu item is updated with the new price.
Call Function(Fail)	If item already exists with the same price or if the item does not exist, the function will return false.

Test-Case Identifier: TC-03
Function Tested: removeFromMenu(string itemName): bool
Pass/Fail Criteria: Test passes if an existing item is removed from the menu list.

Test Procedure	**Expected Results**
Call Function(Pass)	Item is removed from the menu.
Call Function(Fail)	Returns false if the item name is not found in the menu list.

Test-Case Identifier: TC-04
Function Tested: viewOrders(): void
Pass/Fail Criteria: Test passes if the orders are correctly displayed.

Test Procedure	**Expected Results**
Call Function(Pass)	The orders are displayed when the function is called.
Call Function(Fail)	The orders are not displayed when the function is called.

Test-Case Identifier: TC-05
Function Tested: doneOrder(): void
Pass/Fail Criteria: Test passes if order is removed from the order list.

Test Procedure	**Expected Results**
Call Function(Pass)	Order is removed from the order list.
Call Function(Fail)	Order does not get removed from the order list.

6.1.2 Waiter

Test-Case Identifier: TC-06
Function Tested: placeOrder(): bool
Pass/Fail Criteria: Test passes if the order is successfully sent to the chef's order list.

Test Procedure	**Expected Results**
Call Function(Pass)	The order information is sent to the chef's order list and the function returns true.
Call Function(Fail)	The order information did not get sent to the chef's order list and the function returns false.

Test-Case Identifier: TC-07
Function Tested: viewMenu(): void
Pass/Fail criteria: The menu is correctly displayed on the waiter's interface.

Test Procedure	**Expected Results**
Call Function(Pass)	The menu items are correctly displayed on the waiter's interface.
Call Function(Fail)	The menu items are not shown or not shown correctly on the waiters interface.

Test-Case Identifier: TC-08
Function Tested: viewTables(): void
Pass/Fail Criteria: Test passes if the status of each table is displayed on the waiter's interface.

Test Procedure	**Expected Results**
Call Function(Pass)	On the waiter's interface, he/she is shown the status of each table.
Call Function(Fail)	Table status is not shown to the waiter on their interface.

6.1.3 Manager

Test-Case Identifier: TC-09
Function Tested: viewStatistics(): void
Pass/Fail Criteria: Test passes if the statistics are displayed properly on manager's interface.

Test Procedure	**Expected Results**
Call Function(Pass)	Restaurant statistics are properly display on the manager interface.
Call Function(Fail)	Statistics are not displayed or displayed incorrectly.

Test-Case Identifier: TC-10
Function Tested: editBudget():bool
Pass/Fail Criteria: The budget is changed appropriately to show the user's new information.

Test Procedure	**Expected Results**
Call Function(Pass)	The budget is changed and the function returns true.
Call Function(Fail)	The budget is not changed or changed incorrectly and the function returns false.

Test-Case Identifier: TC-11
Function Tested: payroll():void
Pass/Fail Criteria: The payroll is displayed properly on the user's interface.

Test Procedure	**Expected Results**
Call Function(Pass)	The payroll is displayed correctly with the correct information concerning the payroll (employee names, hours worked, etc.)
Call Function(Fail)	The payroll or the information on it is not displayed or displayed incorrectly.

Test-Case Identifier: TC-12
Function Tested: sendPayroll():bool
Pass/Fail Criteria: The payroll is sent to whoever will manage it.

Test Procedure	Expected Results
Call Function(Pass)	The sending process is executed properly and the payroll is sent to the recipient and the function returns true.
Call Function(Fail)	The sending process is not executed properly and the payroll is not sent to the recipient. The function returns false.

6.2 Test Coverage

Our test coverage will include all the major parts of our system, first testing for normal use cases, then moving on to boundary cases. We will also test our user effort estimations to see that they are close to our previous estimations.

For the Manager user, the viewStatistics() test will assure that all appropriate information is sent to the manager's interface. This will cover all functions that collect information, functions that process information, and functions that send the information to the manager interface.

For the Waiter user, the viewMenu() test will cover all functions for user-interface and for accessing menu information. The placeOrder() test will cover all functions relevant to creating an object for the order and sending this order to the chef.

For the Chef user, the addToMenu() test will cover all functions relevant to reading from and writing to the Menu database.

All these tests (and the others above) will provide comprehensive test coverage for all major parts of the system, with special focus paid to the parts most heavily impact the user's experience. Of the 44 proposed functions seen in our class diagram, about 40 of them will be covered; our total test coverage will approach 91%.

6.3 Integration Testing Strategy

Integration Testing will follow something similar to the big-bang approach, testing many parts of our system at once, but not all, using the tests described above. Our approach will focus on the previously described use-cases. By using the big-bang approach on defined and largely closed sections of our system, we increase our ability to localize faults without needing to use much time developing specific module tests.

This will be combined with a bottom-up approach at specific steps only when necessary. For example, Manager function viewStatistics() will be tested before editBudget(), in order to assure that the lower-end case of viewing whatever is on the budget is functioning properly before testing that the budget is able to be changed (and viewed properly).

7. Project Management

7.1 Merging the Contributions from Individual Team Members

As a means of compiling our individual work, we utilized Google Drive as a tool to both communicate with each other and simultaneously work on report components in separate documents and spreadsheets. Some issues that we encountered and were able to overcome were grammatical errors, formatting issues such as varying font and structure, and initial organization errors (fits into the concept of varying font and structure as well). We were able to easily resolve organization and formatting issues by making use of a Table of Contents at the beginning of every component document/report and using a similar template each time. By creating a Table of Contents and distributing work amongst pairings within the group, we were able to find and keep track of our work in the report, allowing us to check easily for grammatical errors and resolve any other human-error associated issues.

7.2 Project Coordination and Progress Report

7.2.1 Weekly Progression

Jan 22 - Jan 29

We formed our group and chose to take on Restaurant Automation as our project focus. We met in person and discussed/brainstormed ideas for each of our three main restaurant components: Food Preparation, Management, and Customer Service. With prior research done both individually and collaboratively on previously submitted Restaurant Automation reports, we were able to devise a proposal including an extensive conceptual definition/pre-layout of our plan of action.

Jan 30 - Feb 19

Utilizing our proposal and ideas we built upon over the course of the few days leading up to the introduction of the first report, we developed a more structured and defined layout for our plan, which included, according to the outline provided to us, a problem statement, functional and nonfunctional system requirements and on-screen appearance requirements, requirements specification (use cases, traceability matrix, and system sequence diagrams), user interface specification, and domain model, as well as a plan of work for each.

Feb 20 - Feb 26

Our group started drafting and completed the first part of the second report. Using our first report, we created UML interaction/sequence diagrams for our fully-dressed use cases, specifically UC-6 (Edit Menu), UC-9 (Order Done), UC-19 (View Statistics), and UC-22 (Payroll Status). A design for each fully-dressed use case implementation (each of the four above) is being worked on currently.

Feb 26 - Mar 5
Our group drafted and completed the second portion of our second report. Continuing what we started in the first part, we focused on two main components: Class Diagram and Interface Specification and System Architecture and System Design. We included a class diagram, list of data types and signatures, network protocol, global control flow, and architectural styles that we employed as a result of weeks of thought and planning.

Mar 6 - Mar 12
During this week, we compiled all of our work and planning from the past three weeks to produce the final version of our second report, for which we focused on algorithms, user interface design and implementation, and design of tests (which included our test cases and integration strategy.

7.2.2 Current Project Status

We are currently working on having an outline/design of the GUI before the demo presentation. Our designs and implementations thus far have been of the following fully-dressed use cases:

1. **UC-6 (Edit Menu)**
2. **UC-9 (Order Done)**
3. **UC-19 (View Statistics)**
4. **UC-22 (Payroll Status)**

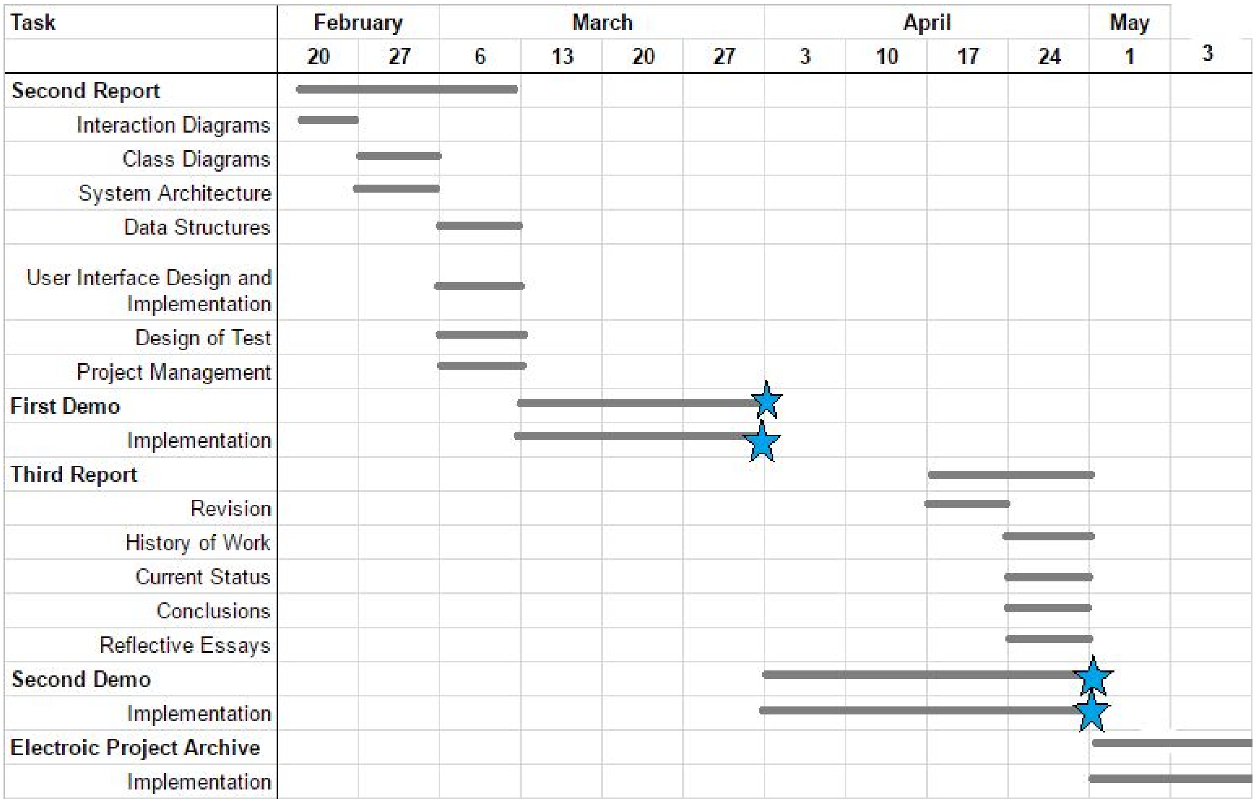
Our system design's login screen has been coded and implemented using Java as shown above, and includes login access for the manager, waiter, chef, and customer, with usernames and passwords already established.

We are working on implementing subfeatures and have provided test cases for all, as shown above as well.

7.2.3 Other Activities

Since our login screen has already been developed, we are working to incorporate the other components (management, customer service, and food preparation) by developing code for those. We have already started planning and organizing these additional subfeatures. All code up to this point has been uploaded in GitHub.

7.3 Plan of Work



7.4 Breakdown of Responsibilities

A detailed examination of the what each member has done so far can be seen on page two of this report. For the remainder of the project, each sub-group will be mainly responsible for their management and implementation. Coordination, however, is key for project coherency, and each sub-group will provide semi-regular updates on their progress; this will be especially true in the weeks leading up to each of the demonstrations.

8. References

1. **2015 Group 3's report for format of document**
   http://eceweb1.rutgers.edu/~marsic/books/SE/projects/Restaurant/2015-g3-report3.pdf

2. **Wikipedia's explanation of interaction diagram**
   https://en.wikipedia.org/wiki/Interaction_overview_diagram

3. **Architectural Styles overview**
   https://msdn.microsoft.com/en-us/library/ee658117.aspx