Report #2:

ChefBoyRD

Restaurant Automation

GROUP #6:

Richard Ahn
Zachary Blanco
Benjamin Chen
Jeffrey Huang
Jarod Morin
Seo Bo Shim
Brandon Smith

GITHUB & WEBSITE:

https://github.com/ZacBlanco/ChefBoyRD
http://blanco.io/ChefBoyRD

SUBMISSION DATE:

March 12, 2017

Individual Contributions Breakdown

All team members contributed equally

Responsibility Matrix									
	Team Member Name								
Task	Possible Points	Richard	Zachary	Benjamin	Jeffrey	Jarod	Seo Bo	Brandon	Completed Points
Sec.1: Interaction Diagrams	30	14%	14%	14%	14%	14%	14%	14%	30
Sec.2: Class Diagrams and InterfaceSpec	10	14%	14%	14%	14%	14%	14%	14%	10
Sec.3: System Arch. and Design	15	14%	14%	14%	14%	14%	14%	14%	15
Sec 4: Algorithms and Data Structures	4	14%	14%	14%	14%	14%	14%	14%	4
Sec.5: UI Design and Implementation	11	14%	14%	14%	14%	14%	14%	14%	11
Sec.6: Design of Tests	12	14%	14%	14%	14%	14%	14%	14%	12
Sec 7: Project Management	18	14%	14%	14%	14%	14%	14%	14%	18
Total Points	100	14.3	14.3	14.3	14.3	14.3	14.3	14.3	100

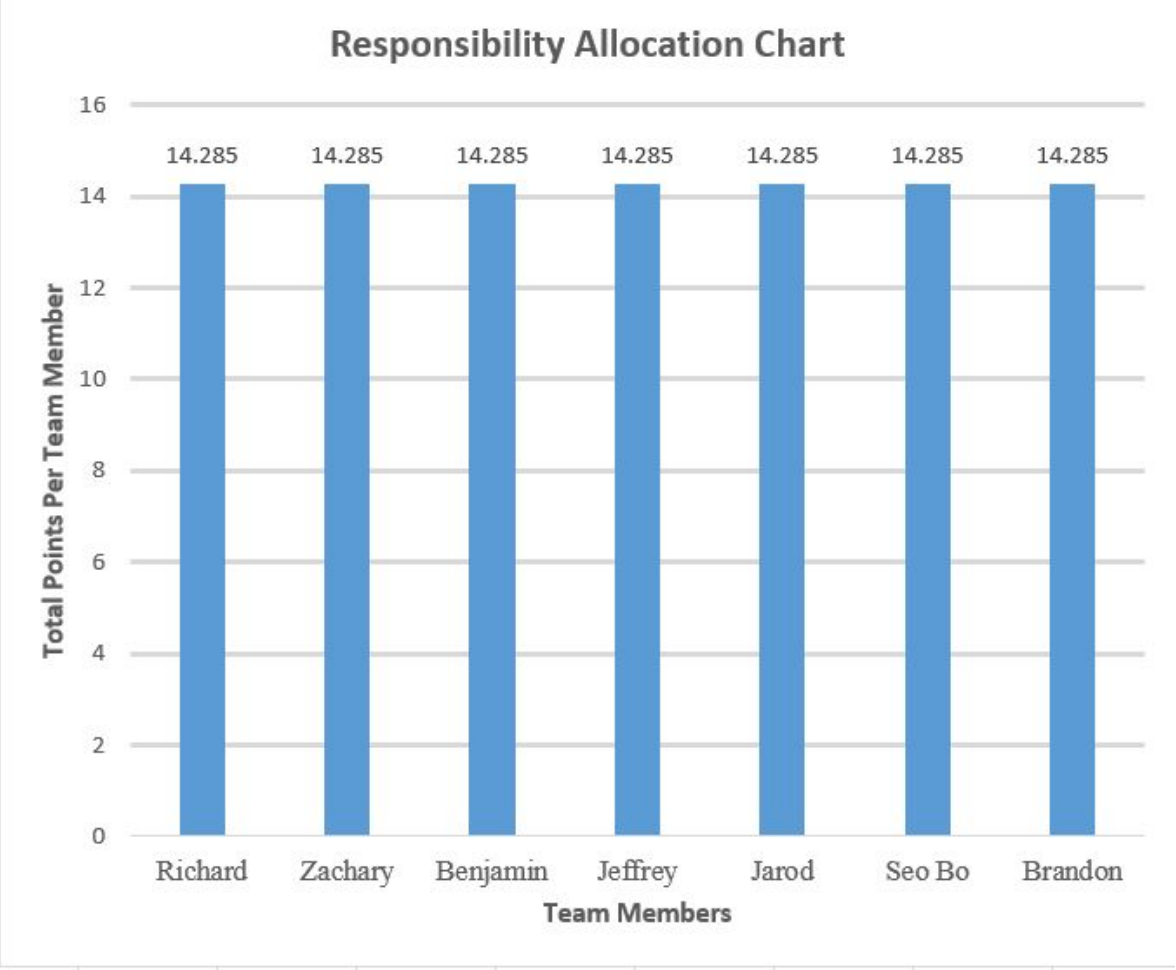
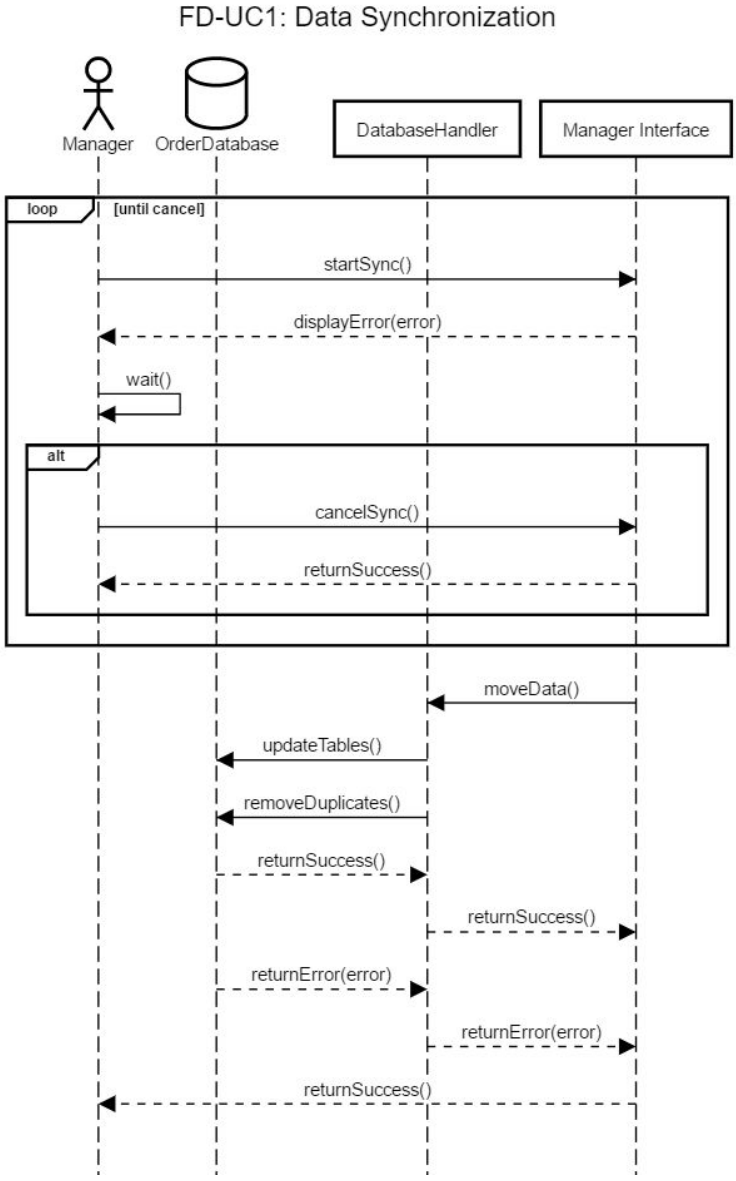


Table of Contents

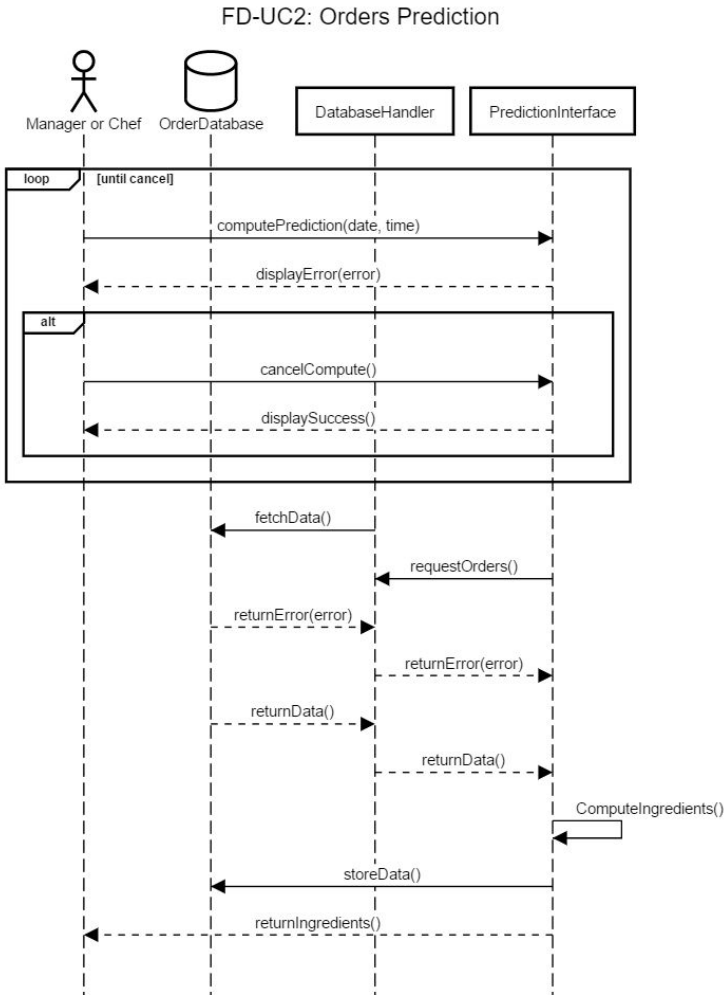
1. Interaction Diagrams

All interaction diagrams are evolved from Report 1, part 3d: System sequence diagrams.

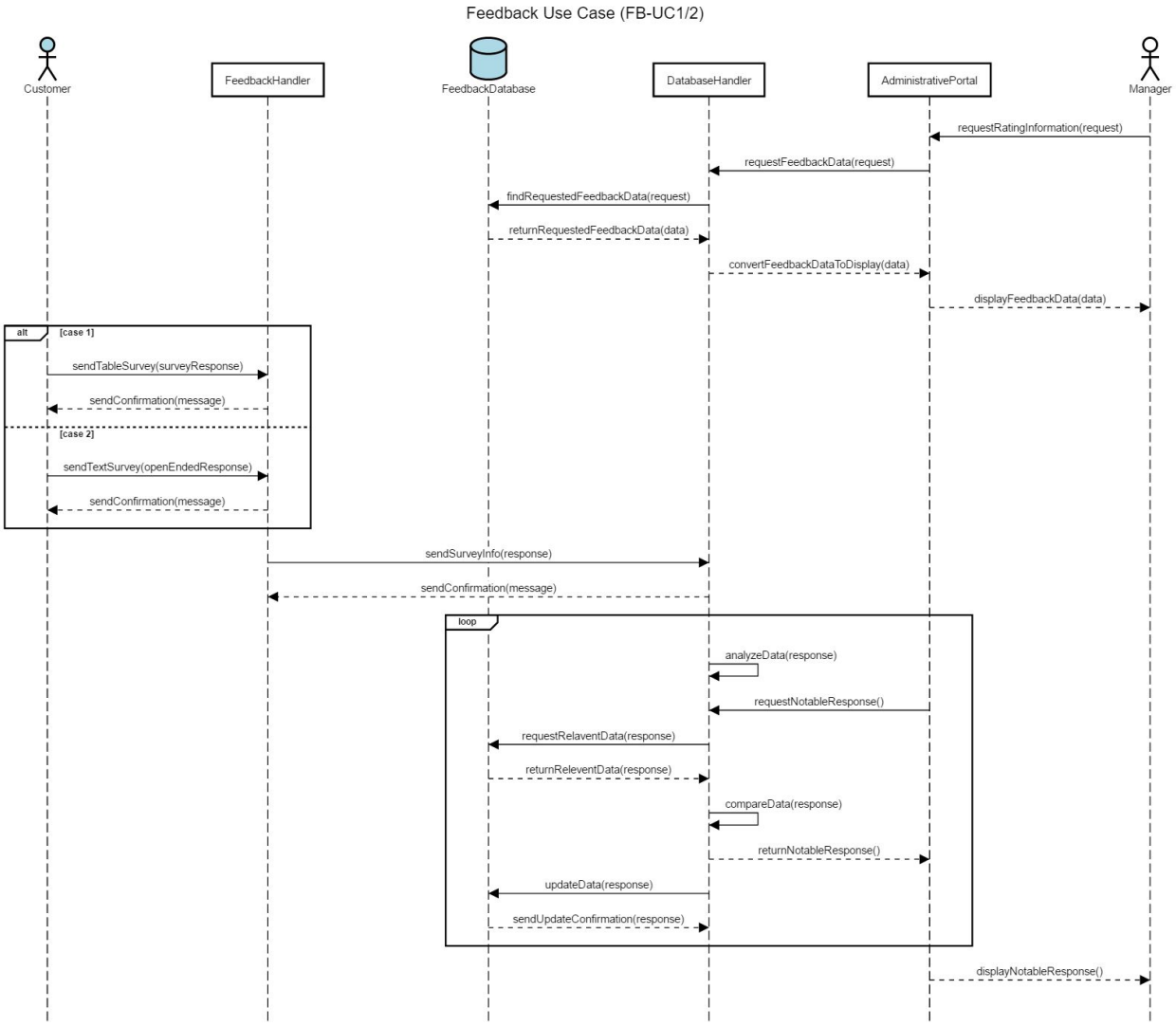


The manager can click a button on the manager's interface to manually update the prediction database with data from the current day. The system automatically performs this, but sometimes we want to update manually. Once the user is successfully authorized the manager interface will attempt to update the tables in the order database. What this does is it asks the DatabaseHandler to move the previous current day's data into the order history to be included in the calculation of the prediction service. If nothing has gone wrong the manager's interface will return success. On an error such as a database error, or authentication issue the interface will

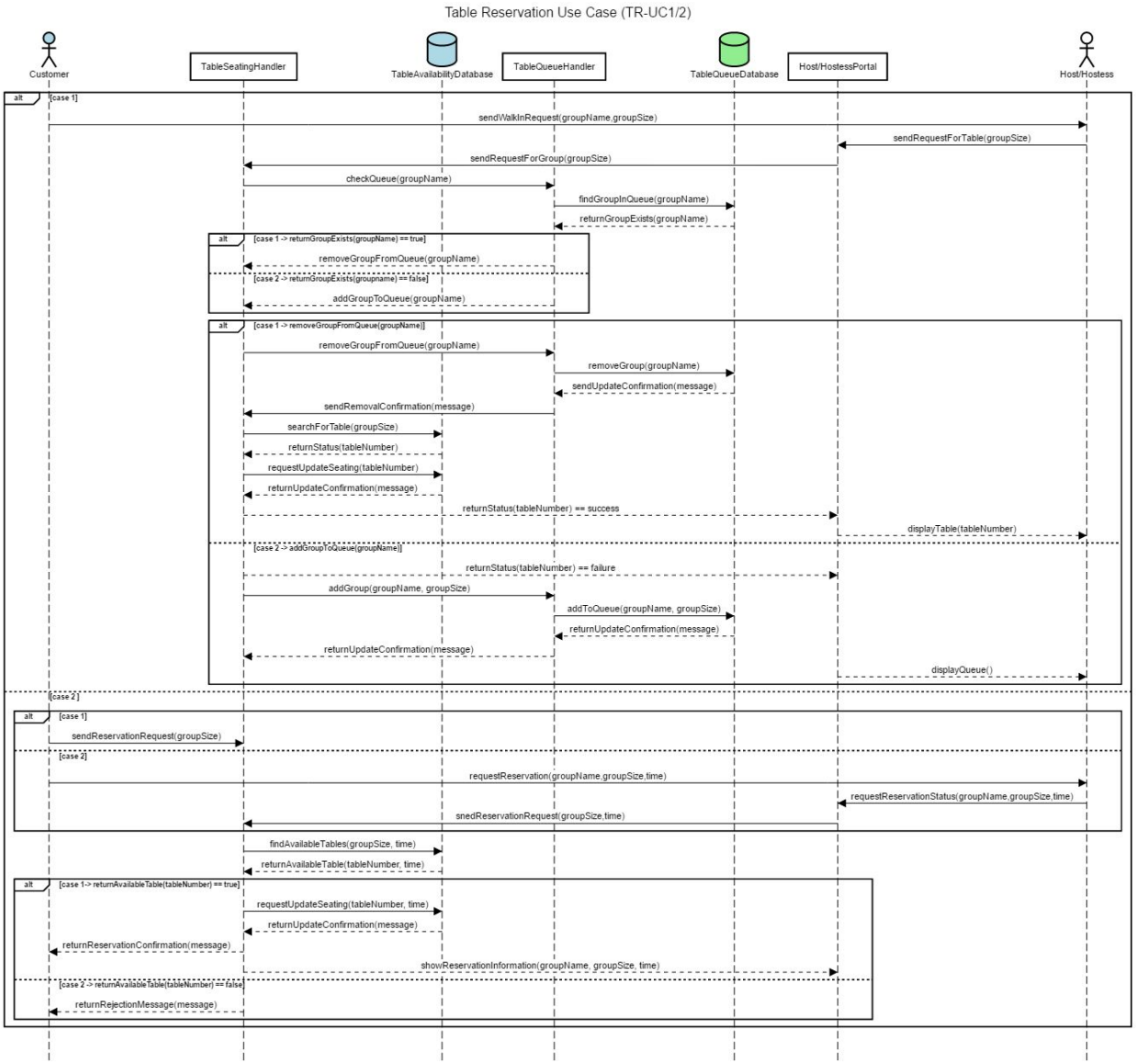
display an error and ask to retry. The Database handler ensures that the interfaces have low coupling.



At the start of the day a manager or chef can open the prediction interface and compute the necessary ingredients that need to be prepared for the current day and time. When this interface is opened, if the model has not computed ingredients for the current day the prediction interface will first attempt to fetch the previous day's data and previous prediction data through the DatabaseHandler. Once that happens the prediction interface will put the data through a model to compute the needed ingredients. The data will then be displayed on the prediction interface and also sent back to the order database to be stored. On a database or time/date error the prediction interface will display the error generated and ask if you want to try again. Because the previous computations are stored in the database this ensures that the prediction interface has high cohesion.



Customers will submit one form or the survey at a time. The feedback handler will determine receive the type of survey that the customer submits and then return a confirmation message that says that the survey information has been received by the system. After the system receives the survey information, the database handler will take care of the analysis of the data as well as any requests that are made by the administrative portal. Once the database handler receives any new information from the feedback handler, the model will then proceed on retrieving any relevant data that is in the database and find if the new data is notable in comparison to the old data. If there is data that is notable, the system will send the information to the administrative portal and display that information for the manager to see. On the other end, the manager can request information from the feedback database to see how a certain subject is performing. The manager will request the information from the administrative portal which will request the information from the database handler. The database handler will then retrieve the relevant information pertaining to the search query. The database handler will then return that information to the portal which will be displayed for the manager to view.



The customer has three different forms of reserving a seat. The three forms are when the customer walks-in and request a table, customer makes a reservation through the online system, and customer makes a reservation through contacting the host/hostess and they input the reservation information. Starting with the walk-in customer, the host will input the information that the customer provides into the host/hostess interface where they will then be able to figure out if there is a table available or the customer will be placed into queue. The reservation system has two cases. One that allows the customer to reserve through the online system or through the host/hostess. Both operations run in a similar manner where there is only one more communication path for the reservation through host/hostess. If the table if available during reservation time, they will be able to reserve the table, otherwise they will get a message saying that their reservation failed.

a. Design Principles

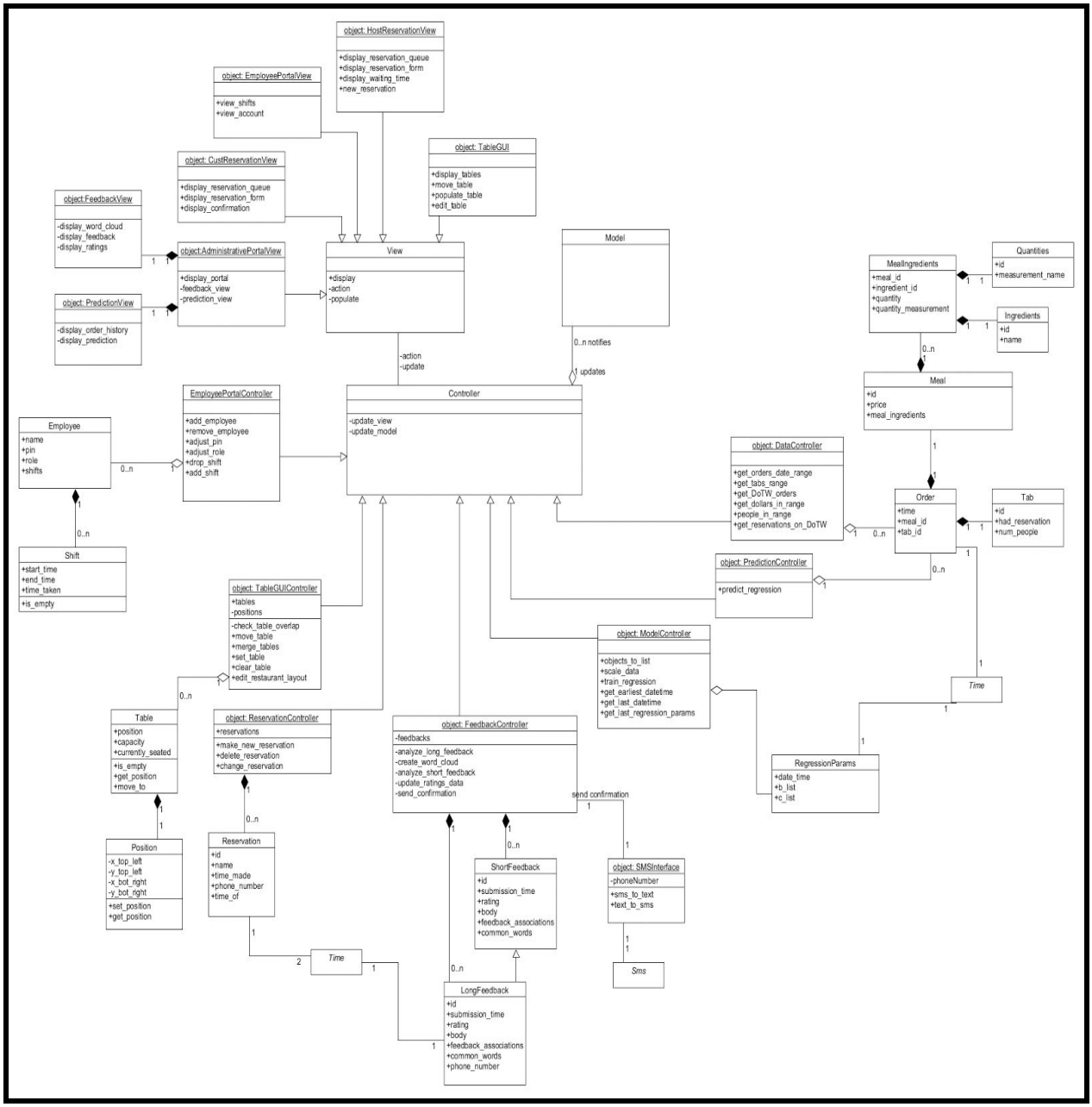
Each of the use cases attempt to limit the number of tasks that each entity will try to handle at any given time. Given that the system is running in continuously in order to keep the databases updated in realtime, the workloads have to be distributed evenly and streamlined so there are no overlapping signals that would cause a conflict in information that is being sent between each handler and database. In order to achieve this, we followed the High Cohesion Principle which places the emphasis on trying to limit the number of computations to prevent a slow response time in any part of the systems. To limit the number of signals being sent between the handlers and databases, we followed the Low Coupling Principle which means that we limited the number of connections that go between each handler and database. The database would go with one handler, and the handler will take care of at most two interactive interfaces used by the employee or customer. As a programmer, we should make the assumption that some customers might not be able to use the program properly, so we limited the amount of information that the customer will provide to prevent a smaller chance of providing the incorrect data. As a result, the Expert Door Principle allows the employee to handle more information than the customer. With these three principles implemented, the strength of each principle will bring forth a robust design that will prevent any miscommunications that will occur within the system.

2. Class Diagram and Interface Specification

a. Class Diagram

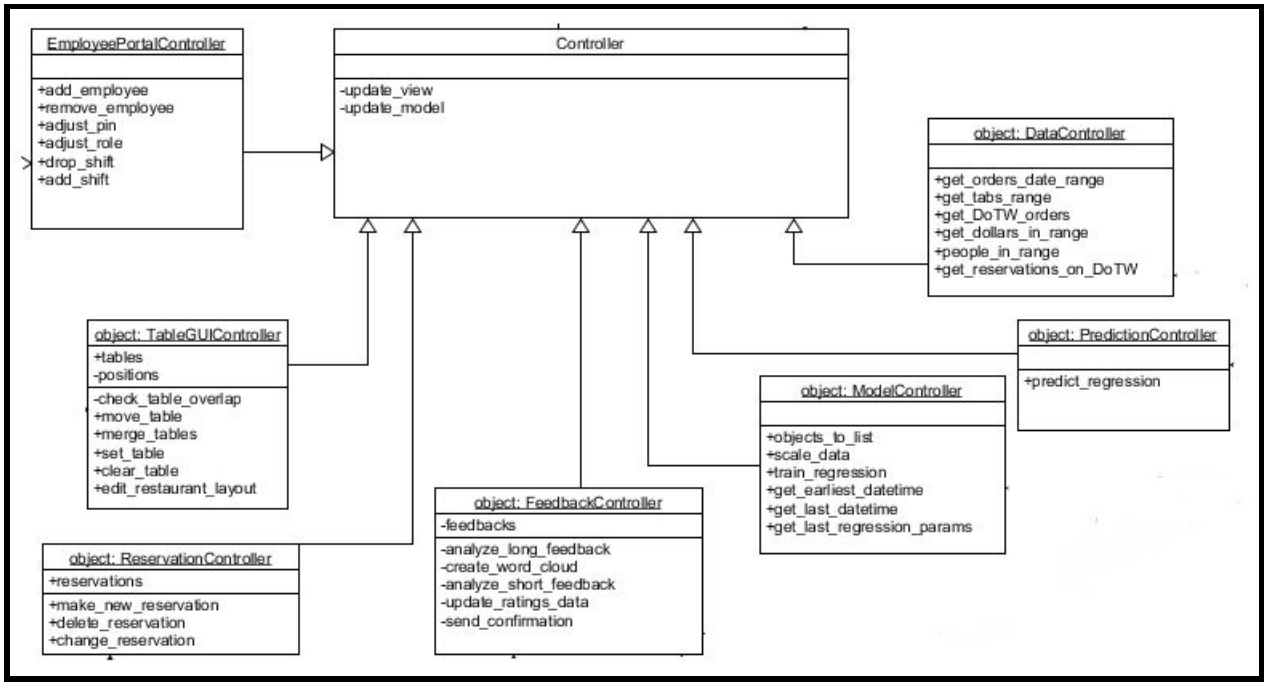
The interaction diagram objects are not exactly the same as the class diagram objects because including each class in the interaction diagram would have complicated the diagram and detracted from the main idea. The mappings from the objects in the interaction diagrams to the class diagram:

Objects in interaction diagram → Classes
ManagerInterface → PredictionView, which is under AdministrativePortalView
DatabaseHandler → is an abstraction for handling data with the database, and each controller uses this abstraction.
PredictionInterface → PredictionView and PredictionController
FeedbackHandler → FeedbackController
TableSeatingHandler → TableGUIController
Host/HostessPortal → TableGUIView and HostReservationView
TableQueueHandler → ReservationController

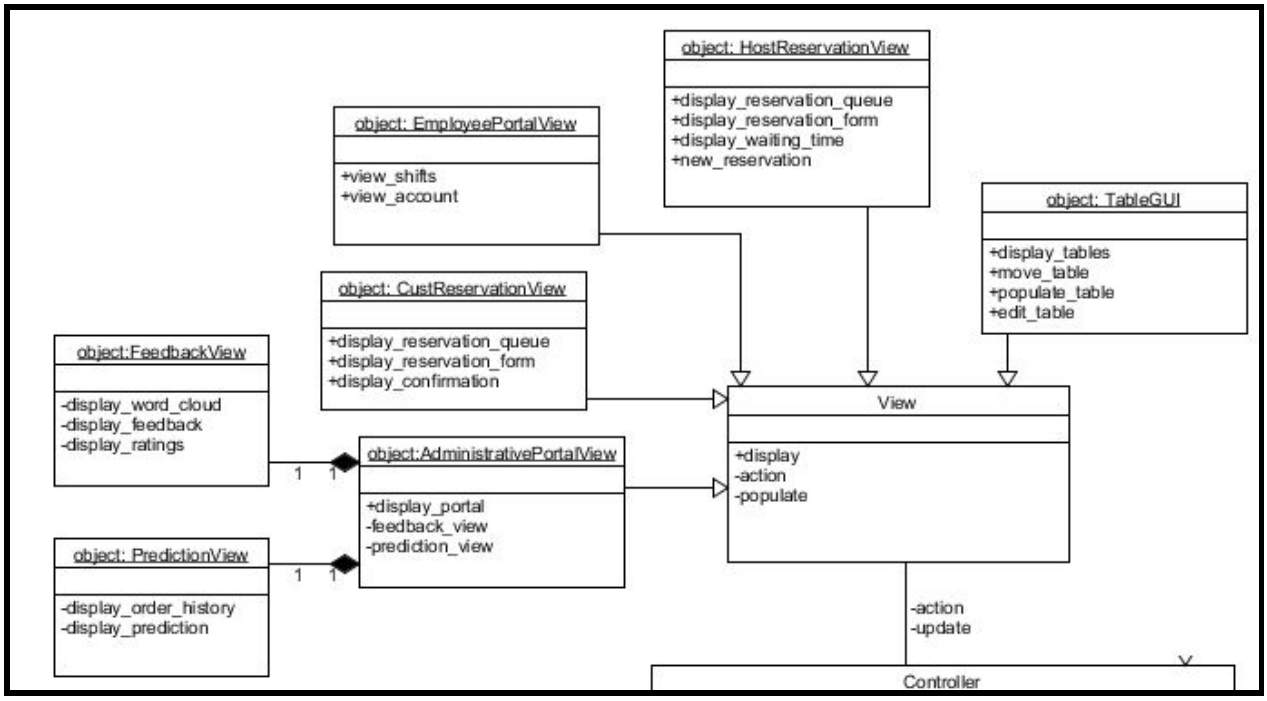


Class Diagram Complete

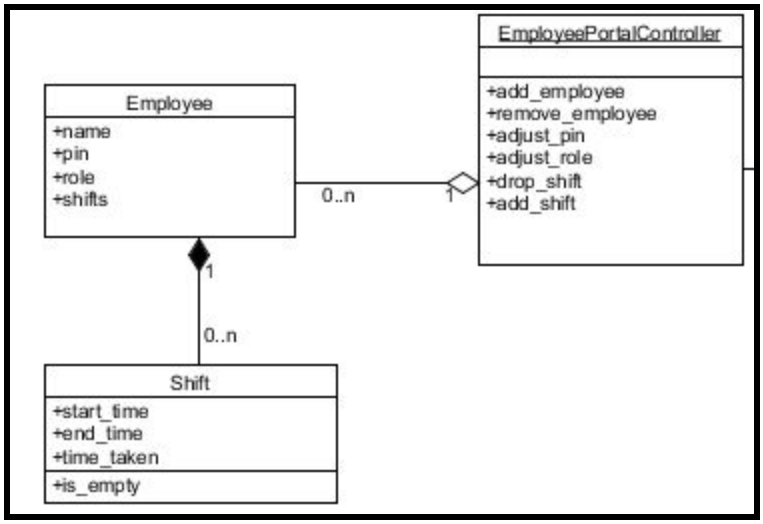
*Abstract Class Time is included twice in diagram for convenience of separation



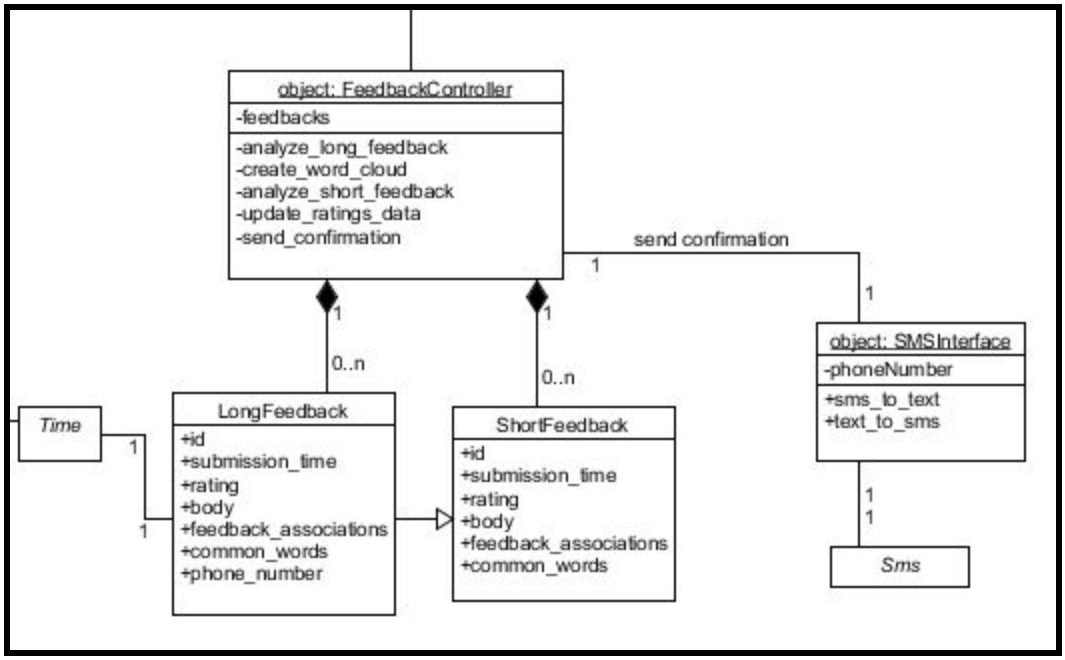
All Controllers



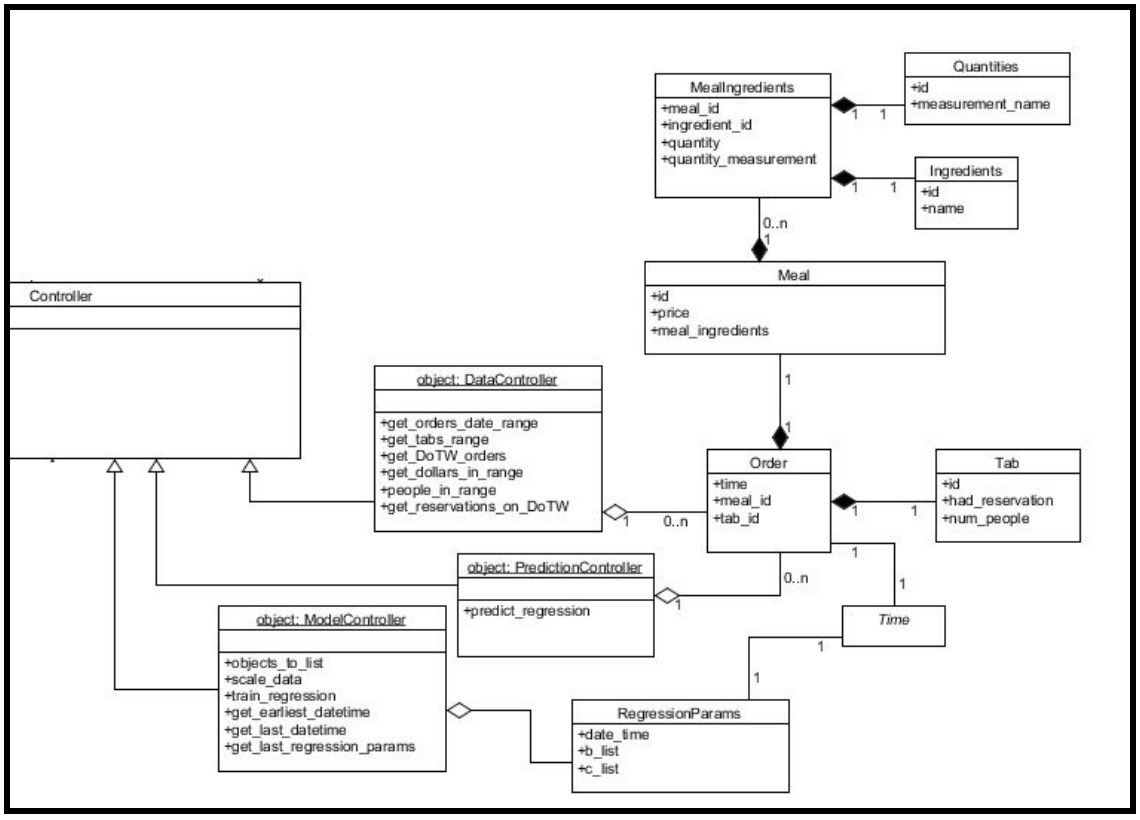
All Views



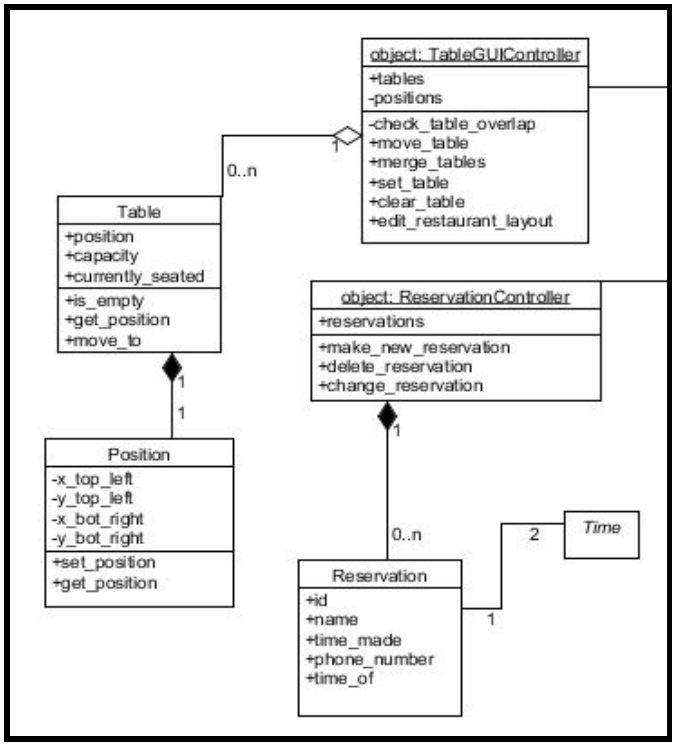
Classes related to EmployeePortalController



Classes related to FeedbackController



Classes related to PredictionController



Classes related to TableGUI and Reservations

b. Data Types and Operation Signatures

Order

This class is used to model customer orders.

Attributes:

+time: Time	A Time class, which refers to the time at which the order was made by the customer
+meal_id: list int	A list of numbers that refers to the IDs of the meal that was ordered
+tab_id: int	A number that refers to the ID of the tab of the order

Meal

This class is used for meals ordered.

Attributes:

+id: int	The number that uniquely identifies the particular meal
+price: int	The price of the particular meal.
+meal_ingredients: List MealIngredients	A list of the mealIngredients objects

MealIngredient

This class represents an ingredient and its quantity, and associates it to which meal this object belongs to. Each meal has several ingredients with a quantity, so each Meal object will have several MealIngredients.

Attributes:

+meal_id: int	The number that uniquely identifies the particular meal that this mealIngredient will be a part of
+ingredient_id: int	The id of the ingredient this mealIngredient reflects
+quantity_measurement: integer	The integer which represent quantity object id, which will describe what the measurement unit is
+quantity: integer	The number of the respective quantity_measurement. How much of that measurement exists in the ingredient

Ingredient

Attributes:

+id: int	The number that uniquely identifies the particular ingredient
+name: String	The unique name of the ingredient

Quantities

This class represents a type of quantity

Attributes:

+id: int	The number that uniquely identifies the particular meal that this mealIngredient will be a part
+measurement_name: String	The name of the measurement quantity. E.g. teaspoons, tablespoons, cups, etc

Tab

This class represents a tab, where there is one per table

Attributes:

+id: int	The number that identifies a unique tab
+had_reservation: bool	True if the table had a reservation. False if table did not have reservation.
+num_people: int	The number of people in the table, where an order was made.

RegressionParams

This class represents the parameters necessary for a regression.

Attributes:

+date_time: Time	The specified time that the regression will range over.
+b_list: List double	Abstract double-precision floating point parameters necessary for the regression model
+c_list: List double	Abstract double-precision floating point parameters necessary for the regression model

PredictionController

This static class is used for controlling the events related to the prediction service.

Attributes:

+most_recent_regression: Time	This time object is used to keep track of when the last regression was done, so the user will know whether the regression is stale or recent.

Methods:

+predict_regression(min: Time, max: Time, modelname: String)	This method is used to do the regression on the order data that was made between the minimum and maximum specified time. The specific regression-model can be specified.

DataController

This static class is responsible for retrieving the data that is stored in the database. This is done by interfacing with the database

Methods:

+get_orders_date_range(min: Time, max:Time)	Get all the orders that were made in the specified time range.
+get_tabs_range(min: Time, max: Time)	Get the tabs for all the orders in the specified time range.
+get_DoTW_orders(min: Time, max: Time)	Get the orders organized into the day of the week in the specified time range.
+get_dollars_in_range(min: Time, max: Time)	Get the dollars that were earned and lost in the specified time range.
+people_in_range(min: Time, max: Time)	Get the people that were working in the specified time range.
+get_reservation_on_DoTW(min: Time, max: Time)	Get all the reservations that occurred organized into the day of the week, in the specified time range.

ModelController

This static class is responsible for modifying the attributes in the models.

Methods:

+objects_to_list(objects: List, cols: List)	Converts the object models into lists that can be compatible with the database
+scale_data(2DList: List, column: int, min: int, max: int)	Scales the data, as more is added to the regression.
+train_regression(2DList: List, StartingParams: List)	This trains the regression, so that it improves with the introduction of new order data.
+get_earliest_datetime()	This gets the earliest regression's date and time
+get_last_datetime()	This gets the last regression's date and time
+get_last_regression_params()	This gets the previous regression's regression parameters

PredictionView

This static class is responsible for creating the display on the Manager and Chef's dashboard, where he or she can view the order predictions of the software.

Methods:

+display_order_history()	This changes the context of the dashboard so that the order history between a specified date range will be displayed in a table
+display_prediction()	This changes the context of the dashboard so that the order prediction is shown.

Table

This class is used for specifying the relevant properties of a table, such as knowing who is seated and it's position.

Attributes:

+position: Position	A Position object that specifies the coordinates of the table object in the restaurant layout
+capacity: int	Integer for max number of people that can fit at a particular table. Differs for different table sizes
+currently_seated: int	Integer for the number of people currently seated at the table

Methods:

+is_empty(): bool	Check the number of people that are currently seated. If 0, then return true.
+get_position(): int	Gets the position of the table in coordinates
+move_to(coordinates: List): void	Moves the table to a specified coordinates

Position

This class is used for specifying the position of the table in x-y coordinates

Attributes:

-x_top_left: int	x coordinate of top-left corner
-y_top_left: int	y coordinate of top-left corner
-x_bot_right: int	x coordinate of bottom-right corner
-y_bot_right: int	y coordinate of bottom-right corner

Methods:

+set_position(xtl: int, ytl: int, xbr: int, ybr: int): void	Takes four parameters and sets these values as the position
+get_position(); int[]	Gets the coordinates of the position in an array

TableGUIController

This class is used for controlling the GUI display

Attributes:

+tables: List Table	A list of tables that are shown in the interface
-Positions: positions	The positions of the tables present in the interface

Methods:

-check_table_overlap(): void	Check if the tables overlap with each other in the graphic user interface
+move_table(table: Table, position: Table.Position): void	This will move the position of the specified table to the specified position
+merge_tables(table1: Table, table2: Table)	This will merge the two specified tables together, creating a new table with a new seating capacity
+set_table(table, Table, numPeople: int): void	This will set the number of people specified in the specified table
+clear_table(table: Table, numPeople: int): void	This will set the number of people at a table to 0
+edit_restaurant_layout(): void	This will allow the restaurant layout to change. Such as adding more tables, removing tables, changing void locations.

TableGUI

This static class is used for seeing the layout of the restaurant and the tables that are present. The intended user, the host, will be able to manipulate the table's position, and the number of people seated

Methods:

+display_tables()	This will change the context of the interface to display the layout of the restaurant with all the tables

+move_table()	This will change the position of the tables, and this method will call the controller to change the position of the table objects
+populate_table()	This will populate the table, based on a specified number of people
+unpopulate_table()	This will remove all people seated at a table visually, and call the controller to update this change.

LongFeedback

This class is used for the long feedback submitted by user via text.

Attributes:

+id: int	Unique integer assigned to each feedback object, to be used as reference
+submission_time: Time	The time the feedback was submitted, or more accurately, when the server was notified of its submission.
+body: String	The body text of the feedback
+phone_number: String	The phone number the feedback was sent from
+feedback_associations: String[]	An array of associations that will be made for each feedback. This will be useful for analyzing the meaning and organization of the feedback. E.g. positive, negative, passive, etc
+common_words: dict (String:int)	A dictionary of common words used in the body of the feedback, ordered from most frequent to least frequent. The frequency of the word will also be stored Depending on user configuration, this list will have a different number of words.

ShortFeedback

This class is used for the short feedback submitted by the user in a survey

Attributes:

+id: int	Unique integer assigned to each feedback object, to be used as reference
+submission_time: Time	The time the feedback was submitted, or more accurately, when the server was notified of its submission.
+body: String	The body text of the feedback. This is optional for short feedback.
+rating: int	The user's rating of the restaurant from 1-5
+feedback_associations: String[]	An array of associations that will be made for each feedback. This will be useful for analyzing the meaning and organization of the feedback. E.g. positive, negative, passive, etc
+common_words: dict (String:int)	A dictionary of common words used in the body of the feedback, ordered from most frequent to least frequent. The frequency of the word will also be stored Depending on user configuration, this list will have a different number of words.

FeedbackController

This class is used for analyzing the feedback that was submitted.

Attributes:

-feedbacks: List Feedbacks	A list of the feedbacks submitted, and requested by the analyzer.
-ratings: List int	A list of the recent ratings that were made using the short feedback.

Methods:

-analyze_feedback(id: Feedback.id)	This method is used for analyzing a particular feedback. What the analysis will do is populate the commonWords, and feedbackAssociations fields of the Feedback object
-create_word_cloud(list: List Feedback,	This method is for generating a word cloud. It

numWords: int, min: Time, max: Time)	will calculate the overall frequency of words used over a period of time specified by min and max. This will cue the View to display the word cloud
-analyze_short_feedback(id: ShortFeedback.id)	This method will analyze the short feedback, or the survey feedback that was submitted. An average customer rating will be specified, and the answers to specified questions will be compiled.
-update_ratings_data()	This method will update the ratings data displayed to reflect the most current ratings
-send_confirmation()	Sends a confirmation text to a customer who sends in LongFeedback

FeedbackView

This static class is used for the manager's dashboard, where they are able to see the feedback that the restaurant received.

Methods:

-display_word_cloud()	This displays a word cloud that is based on the most common words in feedback in a specified time period
-display_feedback()	This will display all the feedback that was sent in, ordered from most recent to least
-display_ratings()	This will display a report on the ratings that customers made using the survey

SMSInterface

This static class is used for interfacing the the Twilio API. This interface is responsible for translating received SMS messages into text for the FeedbackController.

Attributes:

-phoneNumber: String	This is the phone number of the restaurant, needed to receive text messages.

Methods:

-sms_to_text(): Sms	Translates SMS messages into text form, while also taking down the time and the phone number it was sent from.
-text_to_sms(phone_number: String, body: String)	Send a text message to a specified phone number, with the specified message.

Employee

This class is used for the employee object. The details of the employee, such as their name, and role will be included

Attributes:

+name: String	The employee's unique name, which the employee will be referred to as in the system
+pin: int	The employee's unique personal identifcation number, this is used to login.
+role: String	The employee's role. E.g. Waitress, manager, host, etc.
+shifts: List	A list of the employee's work shifts.

Shift

This class is used for specifying a specific work shift

Attributes:

+startTime: Time	The start time of the shift.
+endTime: Time	The ending time of the shift. Cannot be before the start time.
+timeTaken: Time	What time that this particular shift was taken.

Methods:

+isEmpty()	Method used to check if the particular shift has or has not been taken.

EmployeePortalController

This class is used for modifying the employee object, and the shifts associated with that employee.

Methods:

+addEmployee(name: String, pin: int, role: String)	A new employee object will be created with the specified fields. The name, pin, and role.
+removeEmployee(name: String)	Remove an employee object from the database specified by the name.
+adjustRole(name: String, role: String)	Adjust the role of the specified employee to the specified role.
+dropShift(name: String, shift: Shift)	Drop the specified shift object from the specified employee
+addShift(name: String, shift: Shift)	Add the specified shift object to the specified employee

EmployeePortalView

This static class is responsible for creating the employee portal display on the client side.

Methods:

-view_shifts()	This changes the context of the portal so that all the user's shifts will be displayed
-view_account()	This changes the context of the portal so the user's account information will be shown

Reservation

This class is used for the reservation object, which has all the information associated with a reservation

Attributes:

+id: int	The unique reference to a reservation
+name: string	The name of the person who made the reservation
+timeMade: Time	The time the reservation was made

+phone_number:string	The phone number associated with the reservation.
+timeOf: Time	The starting time of the reservation

ReservationController

This class is used for making new reservations and adding this into the database.

Attributes:

+reservations: List Reservations	Lists the most recent reservations, to be shown in the ReservationView

Methods:

+make_new_reservation(name: String, phone_number: String, timeOf: Time)	This will make a new reservation object with the specified fields. The name, date, time, and the phone number.
+delete_reservation(id: int)	This will delete a reservation object that is specified by it's ID
+change_reservation(id: int, name: String, phone_number: String, timeOf: Time)	This will change the reservation specified by the ID. All fields must be changed, if only some of the fields need to be changed, the other parameters can be blank.

CustReservationView

This static class is responsible for being the interface in which the customer can make reservations.

Methods:

-display_reservation_queue()	This displays the current line for the reservation. This is so the customer will know which reservations are open
-display_reservation_form()	This will display the reservation form which will collect the user's information.
-display_confirmation()	This will display a confirmation when a reservation is made.

HostReservationView

This static class is responsible for being the interface in which the host or hostess can manage the reservations

Methods:

-display_reservation_queue()	This displays the current line for the reservation. The host will be able to see each person's contact information
-display_reservation_form()	This will display the reservation form for the host to make reservations for customers
-display_waiting_time()	The waiting time will display, so that the host can notify the customers
+new_reservation()	This action indicates a new reservation has been made and the data will be processed

AdministrativePortalView

This static class is responsible for creating the employee portal display on the client side.

Methods:

+display_portal()	This is to initially display the admin portal
-feedback_view()	This will change the admin portal's context to see the feedback data
-prediction_view()	This will change the admin portal's context to see the prediction data

---

Abstract Classes

Time - A time object that contains the date and time

Sms - A sms object that will refer to all the information in a sms message. This includes phone number, message, time of delivery, etc.

c.Traceability Matrix

	A	B	C	D	E	F	G	H	I	J	K
1		Database	OrderData	AnalyzeData	Survey	FeedbackData	AnalyzeFeedback	Seating	Reservation	ConfirmMessage	EmployeePortal
2	EmployeePortalView										x
3	EmployeePortalController										x
4	Employee										x
5	Shift										x
6	TableGUIController							x			
7	Table							x			
8	Position							x			
9	TableGUI							x			
10	HostReservationView								x		
11	ReservationController								x		
12	Reservation								x		
13	SMSInterface									x	
14	FeedbackView				x	x					
15	ShortFeedback				x	x					
16	FeedbackController					x	x				
17	LongFeedback					x					
18	PredictionController			x							
19	PredictionView			x							
20	DataController	x	x								
21	Order		x								
22	Meal		x								
23	MealIngredient		x								
24	Tab		x								
25	Quantities		x								
26	Ingredients		x								
27	RegressionParams			x							
28	ModelController	x									
29	AdministrativePortalView		x	x		x					
30	CustReservationView								x		

Concepts:

Database

The database will house all of the models in our system. The database can be accessed with a **DataController**, which can make the database queries.

OrderData

The order-data will be accessed by a controller specific for the orders. **DataController** will handle bringing in the order data from an external database will all the order information. The order information will be organized in the following models/classes, **Order, Meal, MealIngredient, Tab, Quantities, Ingredients.** How these classes interact with each other are explained in detail.

AnalyzeData

Analyze the order-data. This is handled by the **PredictionController**, and **PredictionView**. Where the order can be analyzed and viewed by the Manager. This concept is closely related to OrderData, analyzing this data will need the models relevant to the orders.

Survey

The survey is the short form of the feedback. Generally there is a short rating, and this will be represented by a specific class, **ShortFeedback.**

FeedbackData

The feedback is received by the **SMSInterface**, in the form of **LongFeedback**. The SMSInterface translates the SMS abstract class into the feedback class that the system is familiar with. This feedback can also be associated with **ShortFeedback**, from the surveys taken.

AnalyzeFeedback

This data gathered from **SMSInterface** and the Survey are sent to the **FeedbackController** where this controller will analyze this feedback alongside **ShortFeedback**, which is related to the survey. The feedback data eventually news to be displayed to the user, so the **FeedbackView** class is crucial in implementing this.

Seating

The tables are allocated to customers, and a graphic interface is updated to reflect tables with empty seats or full seats. The **TableGUIController** is responsible for making changes to the view, **TableGUI**. The models **Table**, is relevant because these objects must know the number of people seated at a table, and especially it's **Position**, so that the table may be moved around.

Reservation

The reservations are sent in by the customers, being handled by the **ReservationView**. These customers make new reservations, or change their reservations, and the **Reservation** object is changed. This manipulation is carried out by the **ReservationController** class.

ConfirmMessage

Confirmation messages are presented in two ways. They are presented at the Views level, where if the user takes some sort of action, then there is a result and a confirmation. Then there is also a SMS confirmation that can be send out to users that complete a reservation, or submit feedback. This concept is related to all the **Views**, and the **SMSInterface**.

Employee Portal

The employee portal allows general employees of the restaurant to manage their contact information and add or drop work shifts. The **EmployeePortalController** is responsible for making changes to the shift schedule, modifying the **Employee** and **Shift** model. The employee portal interacts with the employees with the **EmployeePortalView**.

3. System Architecture and System Design

a. Architectural Styles

Communication

Using service-oriented architecture(SOA) allows us to create applications and software with known services. We will be using microservices as an implementation of SOA to pass messages and data between applications. Each of our subproblems can be defined as an individual service rather than the entire project as one service. The advantage of this is that each microservice can be built independently of each other, and do not rely on other pages or controllers. If we add or remove services, the existing services would work the same way. Developing in this style will allow for a quick and modular implementation of our service.

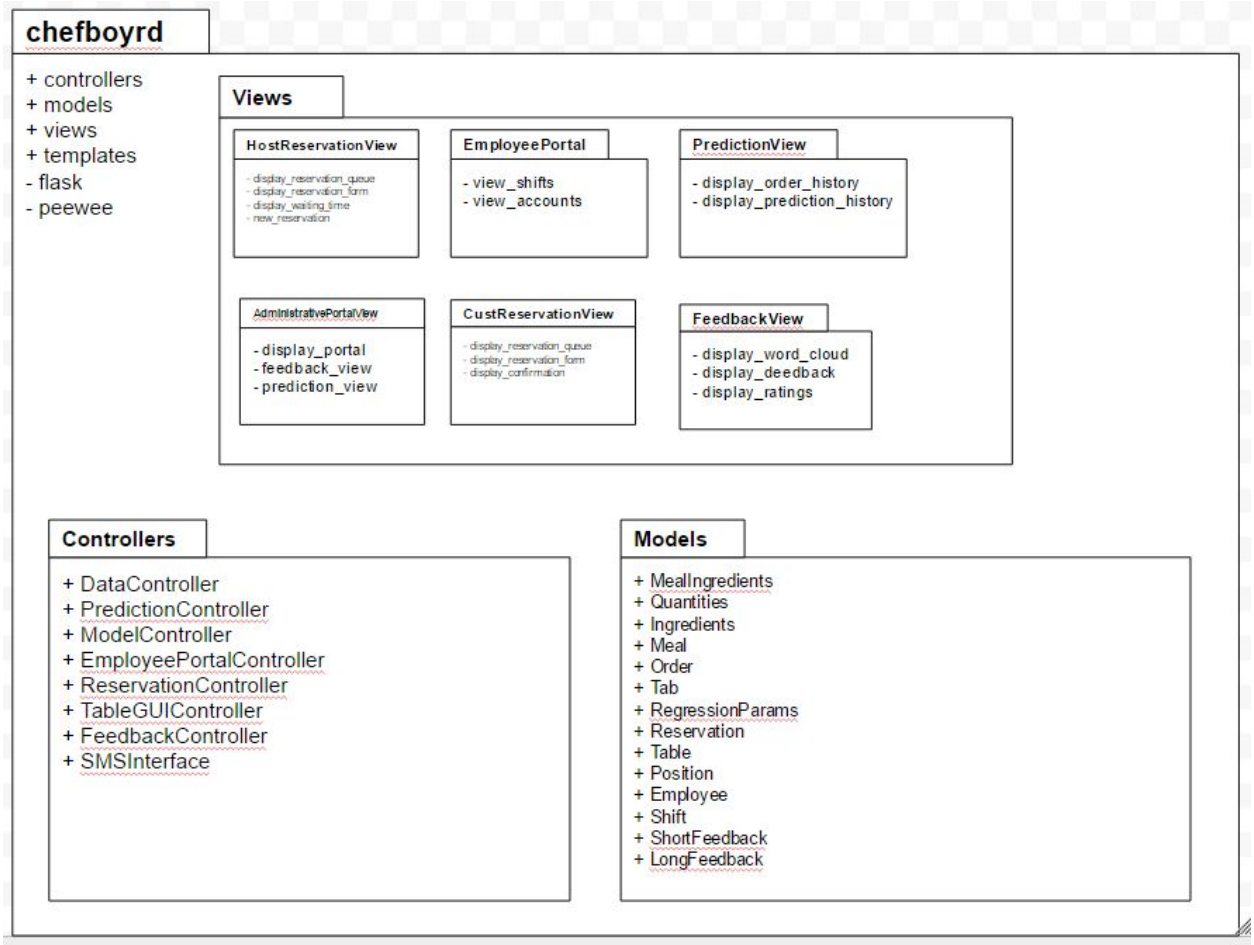
Deployment

We have decided to use a client-server model for our deployment style. In this method the client and server are distinct applications which communicate with each other to solve our problem. In our case the client would be a web browser. The web browser would make requests to the server, and the client would use the interface given by the server to make further requests to the server. With this method of client/server where a web browser is the client we can deploy on multiple platforms and have the interface displayed relatively unchanged opposed to if we developed an app for every device.

Structure

We are employing a Model-View-Controller(MVC) architectural style. In this model the responsibilities are split horizontally into model, view, and controller components. The model component is the central piece which manages data loading and storing of data. The view component is how the resulting data from the model component is displayed to the client. The third component, controller accepts user input, and routes the data to the appropriate controller which handles the logic of our problem domain.

b. Identifying Subsystems



Using the model-view-controller architectural style, our subsystems

c. Mapping Subsystems to Hardware

Because we are using a client-server deployment style our services will need to run on multiple devices. The server will be run on the central computer that the restaurant decides. This server will be the main resource for all the clients, which includes application data and the main database. The client side will be whatever device is needed for each subproblem. Employees use their tablets which will load all data over the web browser provided by the server. Customers can also load the interface, such as the reservation interface through the web browser as well. Because everything is served via web this makes it easy to use basic authentication and security practices to ensure that each role receives the correct interface.

d. Persistent Data Storage

Our system needs to store data for long periods of time to satisfy our subproblems, so persistent data storage is needed here. Everything in our data storage will be stored in databases, this includes order history, inventory, menus, shift information, and our prediction data. Each of these elements will be stored in a separate table in the database, or a separate database altogether. All of this data will be stored using a SQL database. Our architecture affords us the freedom to use any one of MySQL, Postgres, SQLite. We will utilize an ORM (object-relational mapping) layer library called peewee. Peewee stands as a middleware between access to data objects and the actual database. It allows us to use object models within our code and maps them to table relations. The library also abstracts the database connection and querying process to create safer queries. This way our application is protected from SQL injection attacks and our written code is much simpler manner.

e. Network Protocol

Our system sends web pages like many modern applications in order to present information and views to the user. The standard protocol for serving web pages over a network is the HyperText Transfer Protocol (HTTP). Our application allows for the use of HTTPS (Secure HTTP) if the business desires that all traffic be encrypted.

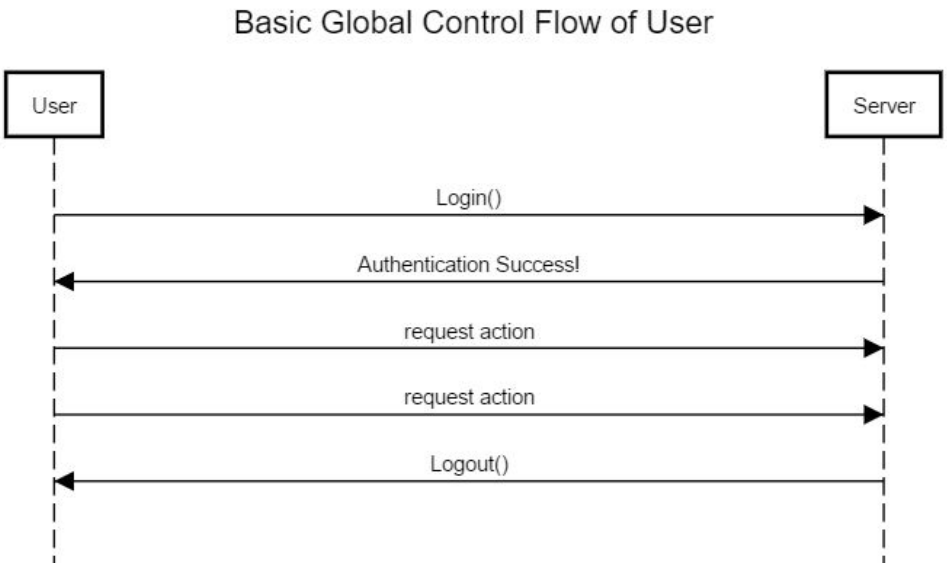
The reason that HTTP(S) was chosen is because it is the standard protocol for sending data and views to the user. Not only has HTTP been implemented across hundreds of platforms but it is a tried-and-true method of interacting with the user that many modern applications use today. The technology is stable and support for the protocol is widespread. Because HTTP is implemented within many cross-platform web browser across almost every operating system it allows our app to become agnostic to the platform that the business chooses to use. The alleviates restrictions on hardware design and lowers the cost to our business customers when implementing the ChefBoyRD system.

f. Global Control Flow

Our system is based off of an internet site architecture. Because of this our application use is event-driven. Every time the user makes a request to the application a thread is created on the web server in order to handle the request, perform any database transactions, or calculations. When the user is not interacting with the application the server remains in an idle state.

Some sequences of actions may need to be performed in a linear fashion in order to complete properly. For example if a host needs to seat a user at a table they would first need to request login → send credentials → navigate to the seating chart → enter the customer information → request seating. This type of operation may need to be performed for the first

time on login. Otherwise the user can just keep making requests in order to fulfill their desired actions. Login and logout should always be the first and last actions performed by the users respectively within the application domain.



The application does not have any components that are dependent on timed actions. We do, however, have an algorithm that uses time explicitly as a datapoint to execute its intended goal of predicting our orders. Our system is solely an event-response type.

Our web application was not designed to take advantage of multiple threads. However, the web server that hosts our page has concurrent multi-threading built-in their service. They use asynchronous threads, each of which handle a request from the threadpool.

g. Hardware Requirements

Server:

Hardware	Minimum Requirements	Recommended Requirements
Operating System	Windows Server 2003/MacOS 10.6.6/Ubuntu or CentOS	Windows Server 2012/MacOS 10.10+/Ubuntu or CentOS
Processor	Intel: Xeon E5502 AMD: Opteron 1352	Intel: Xeon 1230v5 AMD: Opteron 6320
Memory(RAM)	2GB ECC RAM	4GB ECC RAM
Hard Drive	500GB	4TB

Network Card	100/1000Mbps	100/1000Mbps

Client:

Hardware	Minimum Requirements	Recommended Requirements
Operating System	Windows 7/MacOS 10.6.6/iOS 9/Android 5.0.1	Windows 10/MacOS 10.10+/iOS 10/Android 6.1.1
Processor	Intel: i3 2.4Ghz AMD: Phenom X3	Intel: i5 3Ghz AMD: Ryzen 1500X
Display	1280x720 Capable Display	1980x1080 Capable Display
Memory(RAM)	2GB RAM	4GB RAM
Hard Drive	16GB	32GB
Network Card	100/1000Mbps	100/1000Mbps

4. Algorithms and Data Structures

a. Algorithms

One of the main complex algorithms of our application is predicting the quantity of different ingredients being used for a certain time period. This algorithm is composed of multiple steps. First, the data must be retrieved from the database and converted into usable python models. Our database query is formatted such that we get a time-sorted list of raw order data.

Then, we must transform and preprocess the data so that the main mathematical procedure of the algorithm can work efficiently. Our algorithm depends on the day, week, month, year. Therefore, we must extract those values from the time python object. Furthermore, for certain features of our algorithm, we may decide to normalize and scale down its numerical value to a certain range. Next, that data, in the form of a 2D array, will be fed into the algorithm and output its predicted output. We will repeat this algorithm for each ingredient, since the algorithms we are using have only a single output.

For the final and most important part of our algorithm, we will have the user input a specific time range that they want the predicted ingredients for. Since our algorithm only works for a specific time, with the lowest time unit being the hour, we must divide the time range into hour-long buckets and perform the algorithm on each bucket. Finally, we sum up the ingredients used for each bucket, and that is used as our output.

We will run this algorithm using various mathematical models to see if we can find the model that allows us to best predict the restaurant's ingredient usage. In report 1, we outlined and detailed the specific models we would be using. To reiterate, our two main models are the polynomial with a complexity of our choosing and a sinusoidal model.

One issue that we have is that we don't want to train the algorithm every time we want to get the predicted ingredients for a time range. To fix this, we also made the database store our model's parameters so that they can used to easily predict the output for a certain time, since the model is a simple mathematical equation that can be run in constant time.

Within the feedback system, a decision-making algorithm will be used to categorize submissions based on message content. The input will be preprocessed and divided into a list of words which will be compared with various word lists associated with certain sentiments. For example, a list associated with positive feedback would include good, excellent, outstanding, etc. Each matching word will constitute a point in the associated category. In addition, several keywords will indicate what aspect of the experience the feedback relates to, such as service or food. After each category is assessed, the total values will determine how the message is stored.

Various feedback samples will be acquired and tested with our sorting algorithm to determine if they are placed in the appropriate categories. Additional categories and sorting criteria will be explored and added as necessary in response to these tests.

b. Data Structures

Within the database, feedback responses must be stored within categories as determined by the algorithm discussed in the previous section. The messages will be stored inside each category in a data structure. The purpose of the database is to store responses and in a way that is easily accessible. The data will not need to be rearranged or sorted once it is stored, so we decided to use a list (list data structure in Python) to maximize accessibility of elements.

5. User Interface Design and Implementation

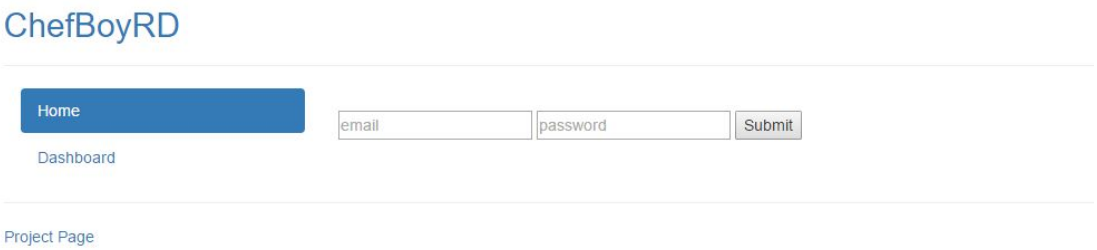
a. User Interface Design

The user interface design will be served via web pages for all users. For the customers trying to make a reservation, the overall interface remains the same. The customer inputs their name, phone number, date and time of the reservation, and party size. The interface is extremely simple to use and displays information in an easy to read fashion. The interface for the employees is also very simple: username and password logins for access and tabs separating the different databases and information.

The GUI design is simplistic and focuses only on necessary information. There are no pictures or unnecessary data to distract the user from his task. On the customer side, the necessary forms and data inputs are large and centered in the screen for easy input and submission. On the employee side, the charts and data are displayed easily and only require a few clicks to access any desired data. Every button and function is labeled and easy to read.

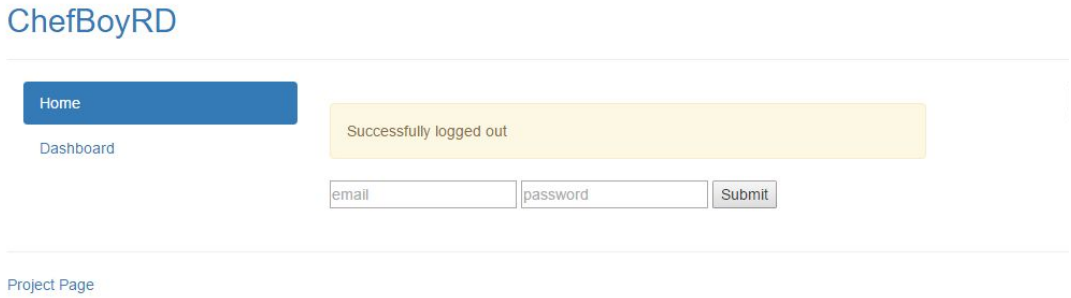
Overall, the user design interface is mostly unchanged. There may be stylistic changes later which means the mockups may not look exactly like the finished interface. However, the overall framework and design of of the employee portal and customer feedback interface will remain unchanged, as the functionally significant design is present in our previous mock-ups in report 1. The following are a couple examples of our implemented User Interface designs:

Home Screen



Login Screen

Welcome Page



Logout Page

b. Implementation

The interface is implemented using an HTML templating system called Jinja2 along with the Bootstrap CSS framework. Jinja works well with the underlying web framework, Flask, that

our application is built upon. Using Jinja allows us to create small UI components from pure HTML and CSS. Then we can make other templates which include the UI components and render information for the user. By having a modular system it allows us to create an easily extensible application where new features and UI components can be added almost effortlessly. The twitter bootstrap significantly simplifies the styling process for our interface as well and allows us to focus on the pure design of the interface rather than the technicalities of UI design such as device scaling and resizing.

6. Design of Tests

a. Test Cases

Test-case Identifier: TC-1 Use Case Tested: FD-UC1, FD-UC2 Pass/fail criteria: The test passes if the day's order data is successfully passed to the database and stored Input data: Current day's order data	
Test Procedure:	Expected Result:
Step 1. Current day's data is sent to the database	System database updates itself with the new data

Test-case Identifier: TC-2 Use Case Tested: FD-UC1 Pass/fail criteria: The test passes if the prediction algorithm produces a prediction of the next day's food usage using order data from the database Input data: Order data from the database	
Test Procedure:	Expected Result:
Step 1. System database performs prediction algorithm on the order data	Prediction algorithm makes a prediction about what food will be used the next day

Test-case Identifier: TC-3 Use Case Tested: FB-UC1 Pass/fail criteria: The test passes if the feedback database receives and stores feedback data, and reports any significant data trends Input data: New customer feedback and existing feedback from the database	
Test Procedure:	Expected Result:

Step 1. Send feedback to the feedback database	Database stores the data and updates database
Step 2. Send a chain of extremely positive feedback regarding an aspect of the service (a particular employee, dish etc)	Database alerts user to the service/employee/dish that received the positive feedback
Step 3. Send a chain of extremely negative feedback regarding an aspect of the service (a particular employee, dish etc)	Database alerts user to the service/employee/dish that received the negative feedback

Test-case Identifier: TC-4 Use Case Tested: FB-UC2 Pass/fail criteria: The test passes if the feedback database receives and stores feedback data, and sorts data based on content Input data: New customer feedback and existing feedback from the database	
Test Procedure:	Expected Result:
Step 1. Send feedback to the feedback database	Database analyzes the feedback and sorts based on content

Test-case Identifier: TC-5 Use Case Tested: TR-UC1 Pass/fail criteria: The test passes if the system correctly determines if a reservation is available for a customer, and makes a reservation if it is available Input data: Customer reservation containing date, time, and party size	
Test Procedure:	Expected Result:
Step 1. Customer requests a reservation, giving the time and party size (assume there is a spot open)	System checks the table schedule and updates to include the new reservation, sends confirmation

Step 2. Customer requests a reservation, giving a time and party size (assume there are no tables free to accommodate the size)	System checks the table schedule and offers an alternate time

Test-case Identifier: TC-6 Use Case Tested: TR-UC2 Pass/fail criteria: The test passes if the system correctly identifies if there is a table for the arriving customer party Input data: Customer party size	
Test Procedure:	Expected Result:
Step 1. Customer party arrives, party size is sent to the system (assume there is a table open)	System checks the table schedule and updates to reflect the new party seating
Step 2. Customer party arrives, party size is sent to the system (assume there are no tables free to accommodate the size)	System checks the table schedule and places customer party into a queue to wait for a table to open

Test-case Identifier: TC-7 Use Case Tested: MS-UC Pass/fail criteria: The test passes if the employee portal successfully provides the employee shift schedule and updates as necessary, and allows the manager to manually change/override schedules Input data: Employee shift data	
Test Procedure:	Expected Result:
Step 1. Employee accesses the employee portal	Portal accurately displays current employee shift schedule
Step 2. Employee makes changes to his/her schedule	Portal updates to reflect these changes
Step 3. Manager accesses the employee portal and creates/edits employee schedules	Portal updates to reflect these changes, overriding any prior employee changes

b. Unit Testing

Our plan for unit testing includes having at least one functional test for each method in every single class of our project. Unit tests will provide at minimum the following inputs for each function in order to minimize realized errors when integrating systems together.

- Check for at least 2 cases of expected input and expected output where the function works properly
- Write at least one test case where the method should accept and invalid input which should raise some kind of error.
- Check edge cases for mathematical functions where boundaries like -1, 0, +1 (and similar) where methods might be expected to go wrong.

By providing these types of tests for every method we can ensure that the method blocks are functional and provide us with satisfactory knowledge that our system shall function correctly when building and integrating our system.

c. Test Coverage

Several of our tests involve state-based testing. For example, the reservation and seating test cases specifically deal with the state of the table schedule (whether there are vacant tables, if the restaurant can accommodate the customer party, etc).

The algorithms we use are tested with sample data to ensure their accuracy. For example, in test case TC-3, we send multiple samples of feedback to the database to test if the algorithm can catch the positive and negative trends we will send. In addition, we will send sample data to the prediction algorithm and feedback sorting algorithm to make sure they process the data and produce their results accordingly.

Overall, the tests cover all aspects of the use cases. Each scenario of each algorithm is tested as well. In addition, since the tests deal with receiving and displaying information from the databases, the tests cover connectivity between the databases and application. Additional tests will be designed and implemented as the need arises.

On top of this we can also utilize code-coverage tools like Coveralls (https://coveralls.io) when building our test system which can help us spot logical areas which we may have missed when designing our unit tests.

d. Integration Testing Strategy

We decided to use bottom-up testing to test our project. Since there are well-defined parts of each component of our project, it would be simple to test each part individually, then test them as they were integrated with each other. For example, the customer feedback portion of our

application is composed of several parts: the customer sending feedback to the system, the database and the algorithm. We test to make sure the feedback is received and stored, then we test the algorithm using the data received.

During project development the developers will write tests to go along with every feature as it is pushed into the main codebase. In order for merges to be able to occur we have set up a continuous integration system using Travis-CI (https://travis-ci.org) and GitHub (https://github.com) which will run the tests with new code changes and requires passing all tests before new changes can be merged in. This ensures that each feature will perform and function without breaking other features that had already been developed.

7. Project Management

a. Merging the Contributions from Individual Team Members

We faced many issues when coordinating the contributions from all team members. At first we all split up section among the group members to contribute to. However, this resulted in low individual accountability, we would rely on each other to take ownership of the report section.

Instead, now we assign a section of the report to one or two individuals, who will be completely responsible for making sure all the requirements of the section are met. This way, if there is some specific information needed for the report, that only a couple team members know, the person responsible for the section will reach out to others.

We used Google Docs, so that we may work simultaneously and stuck to a specific formatting so that we would not create more work for ourselves by continuously formatting report sections. We used Google Drawings for some of the diagrams, but not using a dedicated software also limits us in the tools that we can use. In the case we have specific formatting we need to adhere to, we have one person start the formatting and send the file to the other contributors.

After the initial hurdles, we figured out what works for our team and our coordination improved.

b. Project Coordination and Progress Report

Use Cases finished:

- none

Use Cases in progress:

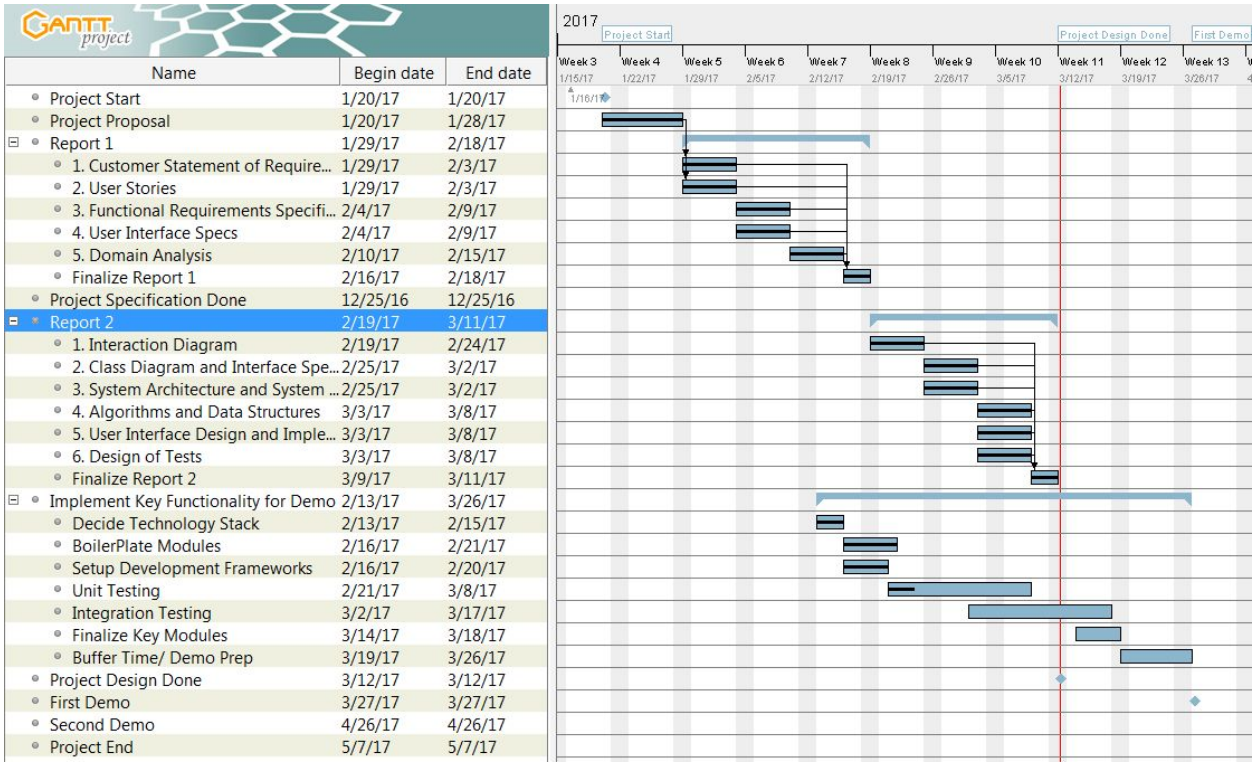
- FB-UC1
    - Set up basic Twilio example. Can send SMS to server, store plaintext of SMS, and message sender can get a message response. Access contents of individual messages for later use in sorting algorithm.
- FD-UC1/2, TR-UC1/2
    - Set up a web page where user can enter credentials and access all their appropriate interfaces
- FD-UC1
    - Specifying the prediction algorithm that will be used.

Functional components:

- Authentication Module for different user roles
- Twilio basic SMS interface, set up on Virtual Private Server (VPS)
- MVC template set up on Flask

- Basic index page + Bootstrap CSS for our website
- Login/Logout Pages

c. Plan of Work



Name	Begin date	End date
Project Start	1/20/17	1/20/17
Project Proposal	1/20/17	1/28/17
Report 1	1/29/17	2/18/17
1. Customer Statement of Require...	1/29/17	2/3/17
2. User Stories	1/29/17	2/3/17
3. Functional Requirements Specifi...	2/4/17	2/9/17
4. User Interface Specs	2/4/17	2/9/17
5. Domain Analysis	2/10/17	2/15/17
Finalize Report 1	2/16/17	2/18/17
Project Specification Done	12/25/16	12/25/16
Report 2	2/19/17	3/11/17
1. Interaction Diagram	2/19/17	2/24/17
2. Class Diagram and Interface Spe...	2/25/17	3/2/17
3. System Architecture and System ...	2/25/17	3/2/17
4. Algorithms and Data Structures	3/3/17	3/8/17
5. User Interface Design and Imple...	3/3/17	3/8/17
6. Design of Tests	3/3/17	3/8/17
Finalize Report 2	3/9/17	3/11/17
Implement Key Functionality for Demo	2/13/17	3/26/17
Decide Technology Stack	2/13/17	2/15/17
BoilerPlate Modules	2/16/17	2/21/17
Setup Development Frameworks	2/16/17	2/20/17
Unit Testing	2/21/17	3/8/17
Integration Testing	3/2/17	3/17/17
Finalize Key Modules	3/14/17	3/18/17
Buffer Time/ Demo Prep	3/19/17	3/26/17
Project Design Done	3/12/17	3/12/17
First Demo	3/27/17	3/27/17
Second Demo	4/26/17	4/26/17
Project End	5/7/17	5/7/17

d. Breakdown of Responsibilities

Subteam A - Richard and Zac

Subteam B - Ben, Seo Bo, and Jarod

Subteam C - Brandon, Jeffrey

Module/Class*	**Team Member Responsible**
PredictionView	Zac
PredictionController	Richard
OrdersController	Subteam A
AdministrativePortalView	Subteam A + B
FeedbackForm	Subteam B
FeedbackReview	Subteam B
FeedbackResponseAnalyzer	Jarod
SmsInterface	Seobo
HostView	Subteam C
ReservationController	Subteam C

TableGUIController	Subteam C
EmployeePortalView	Subteam C
DatabaseHandler	Subteam A + B + C
AuthHandler	Zac

*some of these classes require implementing models, most of which are exclusive to a class, so the do not need to be mentioned here. See Class diagrams for in-depth information.

Tests will exist for all classes within the project. The author of each class is responsible for writing the tests as well. Unit and integration tests will be written to ensure that the project can function in pieces and as a whole together. When merging code into the master branch of the codebase we will at that point, ensure that tests have been written and pass before merging.

Builds and testing will be performed by Travis-CI. It is a tool which integrates with Github to run unit tests and integration tests as new modules are added to project branches. It helps determine the commits, and thus code changes, which result in failing unit and integration tests. This will enable us to pinpoint issues swiftly and with high accuracy. It will also help prevent buggy code from entering the codebase.

Integration into the main branch will be done by each group member as they finalize their modules/classes. It will be coordinated by:

- Seobo
- Zac

8. References

- Plan of Work done using GanttProject (http://www.ganttproject.biz/)
- Interaction Diagrams drawn using SequenceDiagram (http://sequencediagram.org)
- UML Class Diagrams done using UMLlet Software (http://www.umlet.com/)
- Class Diagram symbols used following: Russ Miles and Kim Hamilton: *Learning UML 2.0* Reilly Media, Inc. 2006.
- Jinja2 Templating System for Python (http://jinja.pocoo.org/docs/2.9/)
- Flask Web Framework for Python (http://flask.pocoo.org/docs/0.12/)
- Bootstrap CSS Framework (http://getbootstrap.com/)
- Unit test and integration testing: Travis-CI (https://travis-ci.org)
- Version control and project management: Github (https://github.com)
- Coveralls (https://coveralls.io)
- Reference Restaurant Data Model (http://www.databaseanswers.org/data_models/restaurant_bookings/index.htm)