TurboYums

Restaurant Automation Software Suite

https://fastfoood.github.io/SEWebsite/

Group 13

Second Report

Michelle Curreri
Suvranil Ghosh
Ziad Mallah
Anthony Merheb
Roshni Shah
Holly Smith
Hersh Shrivastava
Brandon Tong
Dante Torello

Table of Contents

Individual Contributions

Topics	Michelle	Suvranil	Ziad	Anthony	Roshni	Holly	Hersh	Brandon	Dante
Interaction Diagrams	22.22%	11.11%	11.11%	11.11%	16.5%	0%	0%	16.5%	11.11%
Class Diagram	0%	15%	37%	5%	0%	0%	0%	0%	43%
Data Types and Operation Signatures	16.67%	11.11%	11.11%	10%	11.11%	11.11%	6.66%	11.11%	11.11%
Traceability Matrix	0%	0%	0%	100%	0%	0%	0%	0%	0%
Architectural Styles	25%	5%	0%	0%	35%	0%	35%	0%	0%
Identifying Subsystems	55%	0%	0%	0%	0%	0%	0%	45%	0%
Mapping subsystems to Hardware	0%	0%	20%	0%	0%	80%	0%	0%	0%
Persistent Data Storage	40%	0%	40%	0%	0%	0%	0%	10%	10%
Network Protocol	0%	0%	25%	0%	75%	0%	0%	0%	0%
Global Control Flow	0%	0%	0%	0%	75%	0%	0%	0%	25%
Hardware Requirements	0%	10%	0%	0%	0%	67.5%	0%	0%	22.5%
Algorithms	0%	0%	25%	0%	25%	25%	25%	0%	0%
Data Structures	15.66%	0%	33.33%	15.66%	33.33%	0%	0%	0%	0%
User Interface Design and Implementation	0%	90%	0%	10%	0%	0%	0%	0%	0%
Test Cases	26.33%	0%	0%	7%	10%	0%	0%	23.33%	33.33%
Test Coverage	0%	0%	0%	60%	0%	30%	10%	0%	0%

Integration Testing	10%	0%	0%	0%	10%	0%	0%	0%	80%
Merging the Contributions from Individual Team Members	90%	0%	10%	0%	0%	0%	0%	0%	0%
Project Coordination and Progress Report	0%	0%	0%	40%	10%	0%	0%	25%	25%
Plan of Work	10%	0%	0%	10%	10%	0%	0%	70%	0%
Breakdown of Responsibilities	15%	0%	0%	20%	20%	8%	0%	9%	33%
References	85%	0%	0%	0%	0%	0%	0%	15%	0%
Editing	10%	20%	20%	10%	20%	0%	0%	0%	20%
Organization	15%	20%	15%	15%	10%	10%	0%	0%	15%
Table of Contents	0%	0%	0%	0%	100%	0%	0%	0%	0%

1. Interaction Diagrams

UC-2: Payment

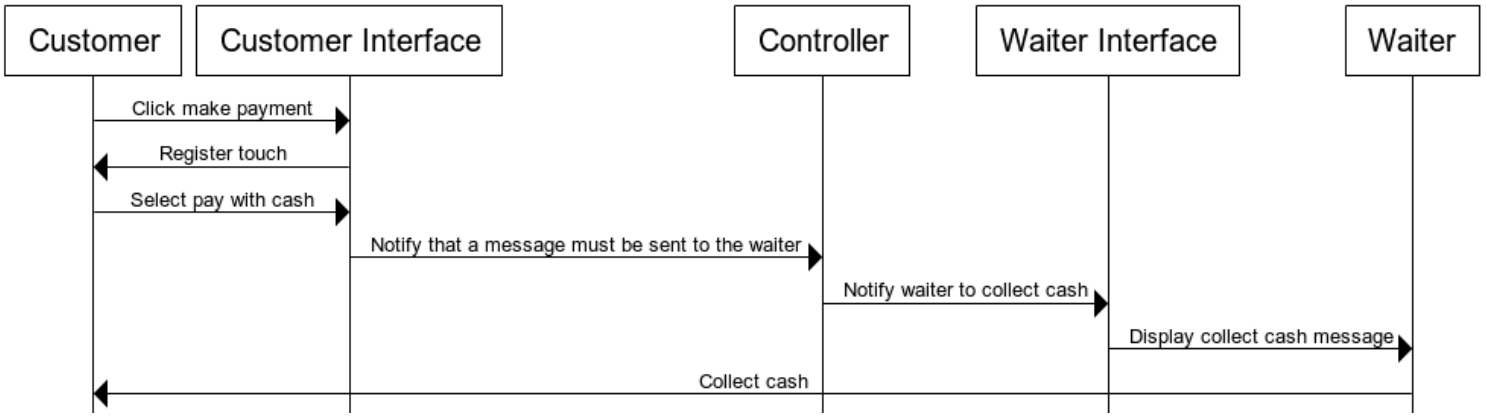


Figure 1: Payment with Cash

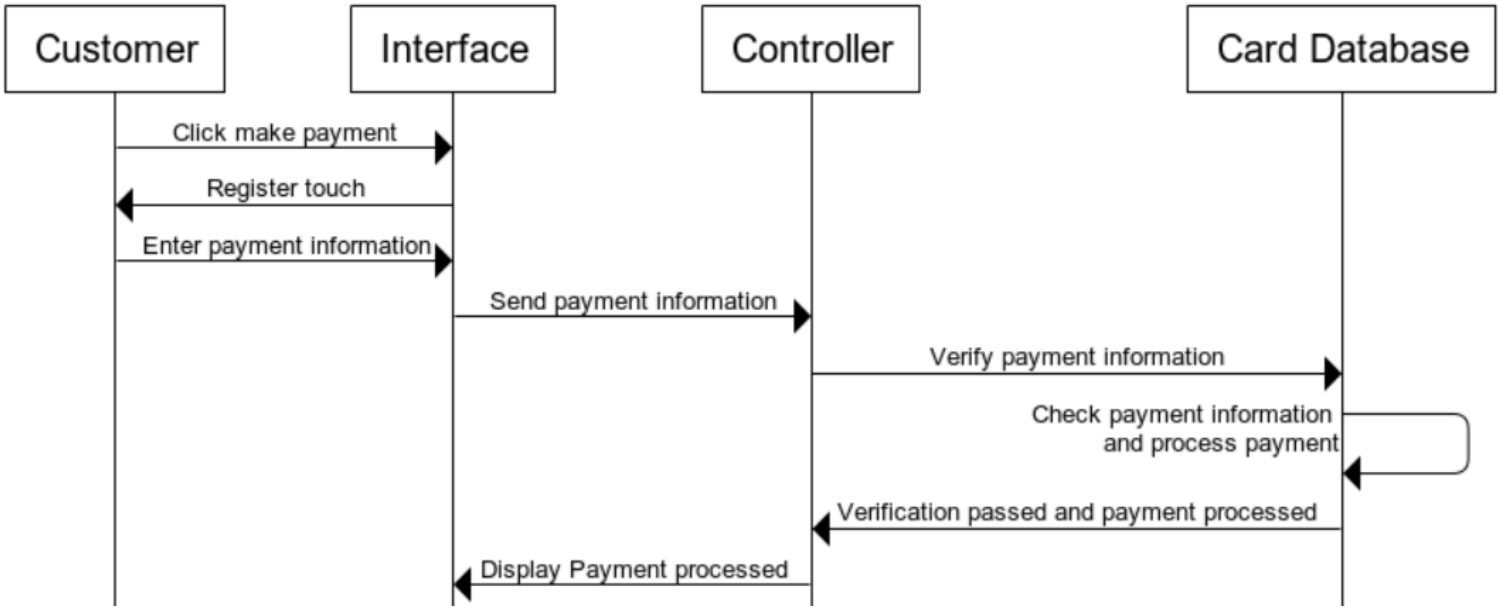


Figure 2: Payment with Card Verification Successful on First Try

These sequence diagrams display some of the potential paths that could be taken when attempting to pay utilizing the TurboYums payment processing. The second diagram shows a best case scenario while attempting to pay for a meal with a credit or debit card, after the customer interacts with the interface and enters their payment information, the information will be sent to the controller, which will then send the data to the specific database for the card (for e.g.: VISA, American Express, MasterCard etc.) in order to verify that the information sent is in fact a valid. The second diagram shows this process running smoothly with valid card information being entered on the first try. The third diagram shows a similar process but with invalid card information being entered, which results in a loop until the user enters valid information that may be used for payment.

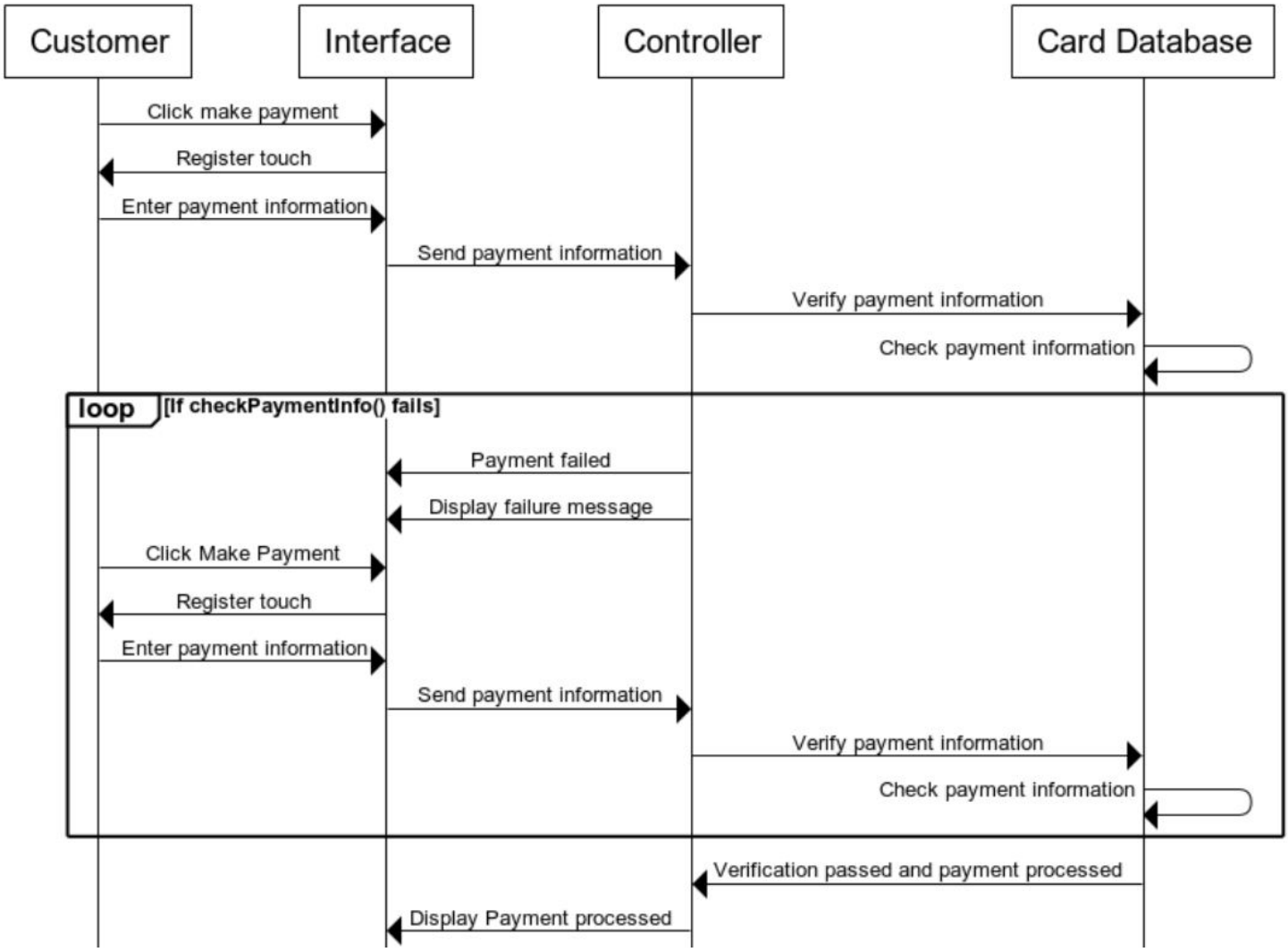


Figure 3: Payment with Card Verification Failure

The first diagram shows what would happen if a customer attempts to pay with cash, which results in additional involvement of a waiter since a waiter must be notified of the payment in order to pick the cash up off of the table.

The high cohesion and low coupling principles were heavily utilized in various different locations while creating the diagrams. The employment of a controller utilizes the high cohesion principle because it allows many of the other objects, such as the interfaces and database to perform their intended tasks (interacting with the user and checking the card information) without having to send messages to other objects involved in the diagram about the status of their actions. Instead of the interface sending the card information to the database, or the customer interface sending information to the waiter interface that cash must be received, the interfaces send this information to the controller so that the controller may then perform the appropriate tasks as needed. The low coupling principle was largely utilized in the cash payment process in order to keep things quick and clean. Information is telephoned down the line from one object to another, through small connections and links, especially between the controller, waiter interface and waiter, this end of the diagram is mostly singly linked objects.

The expert doer was most used for the interfaces and the databases because these objects are the only ones that can successfully perform the tasks that they are undergoing. The interface must serve as the bridge

between the user and the software, and the card database is the only thing that can be used to verify the card information.

UC-3: View Menu

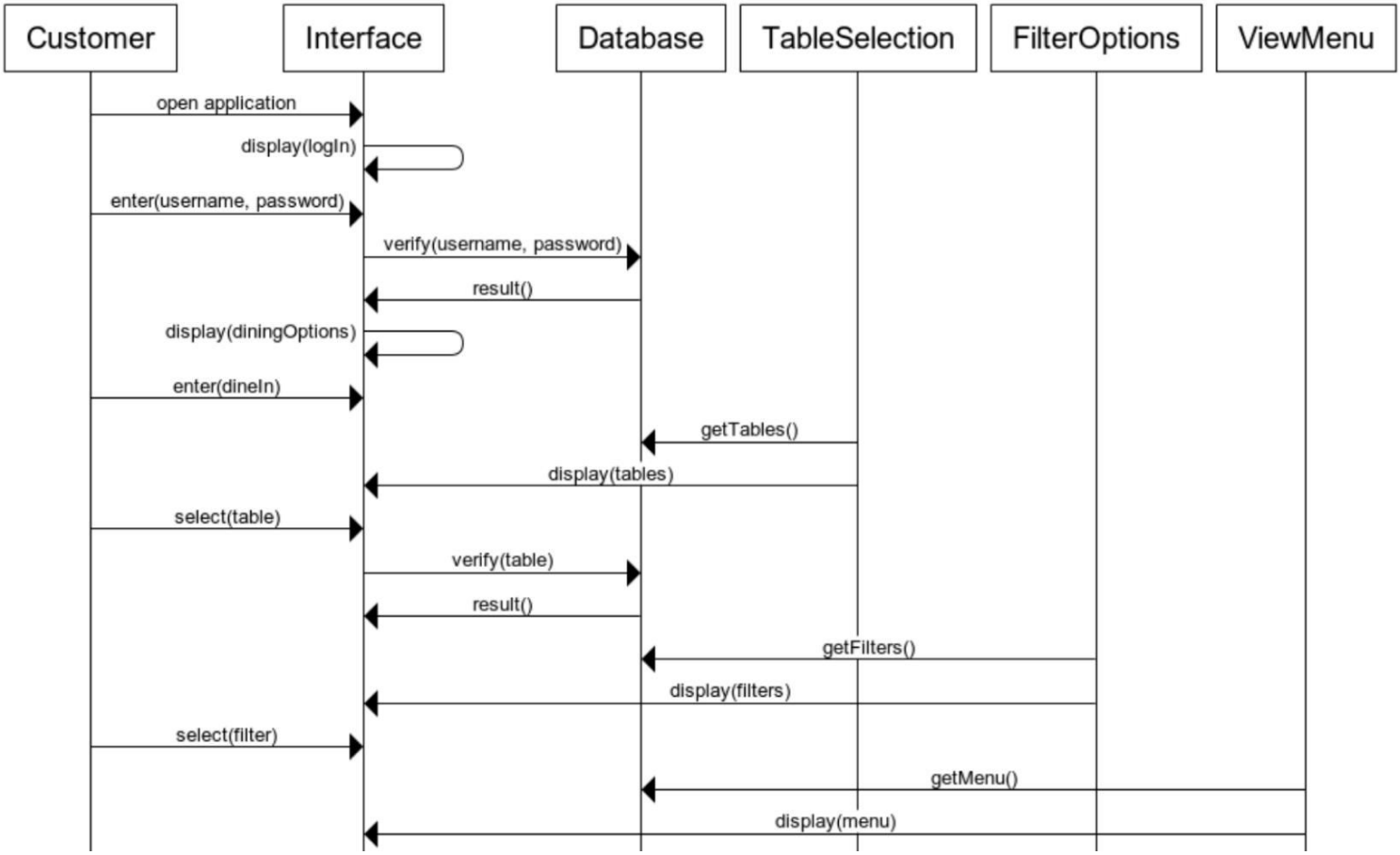


Figure 4: View Menu with Dine-In Selection

One design principle employed in this sequence is the High Cohesion Principle. The High Cohesion Principle says that an object should not take on too many computational responsibilities. This principle is utilized in that each class only takes on responsibilities that have to do with its specific functionalities and doesn't take on more than it can handle at a time.

Another design principle used was the Expert Doer Principle, which says that each class is an expert in a specific function. This is shown in the diagram because, as an example, the only function for TableSelection is simply to display the table layout and have the guest select a table. After this function is over, responsibility is passed on to FilterOptions, where the sequence continues on.

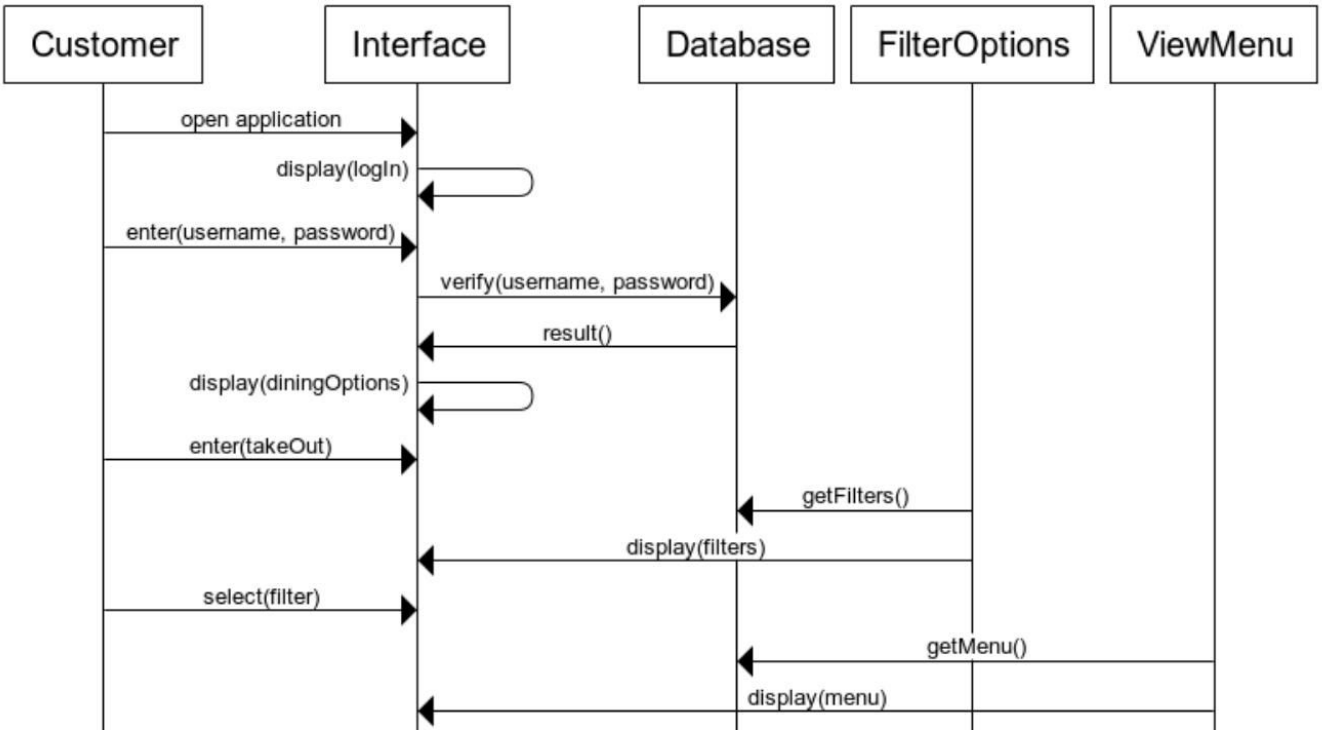
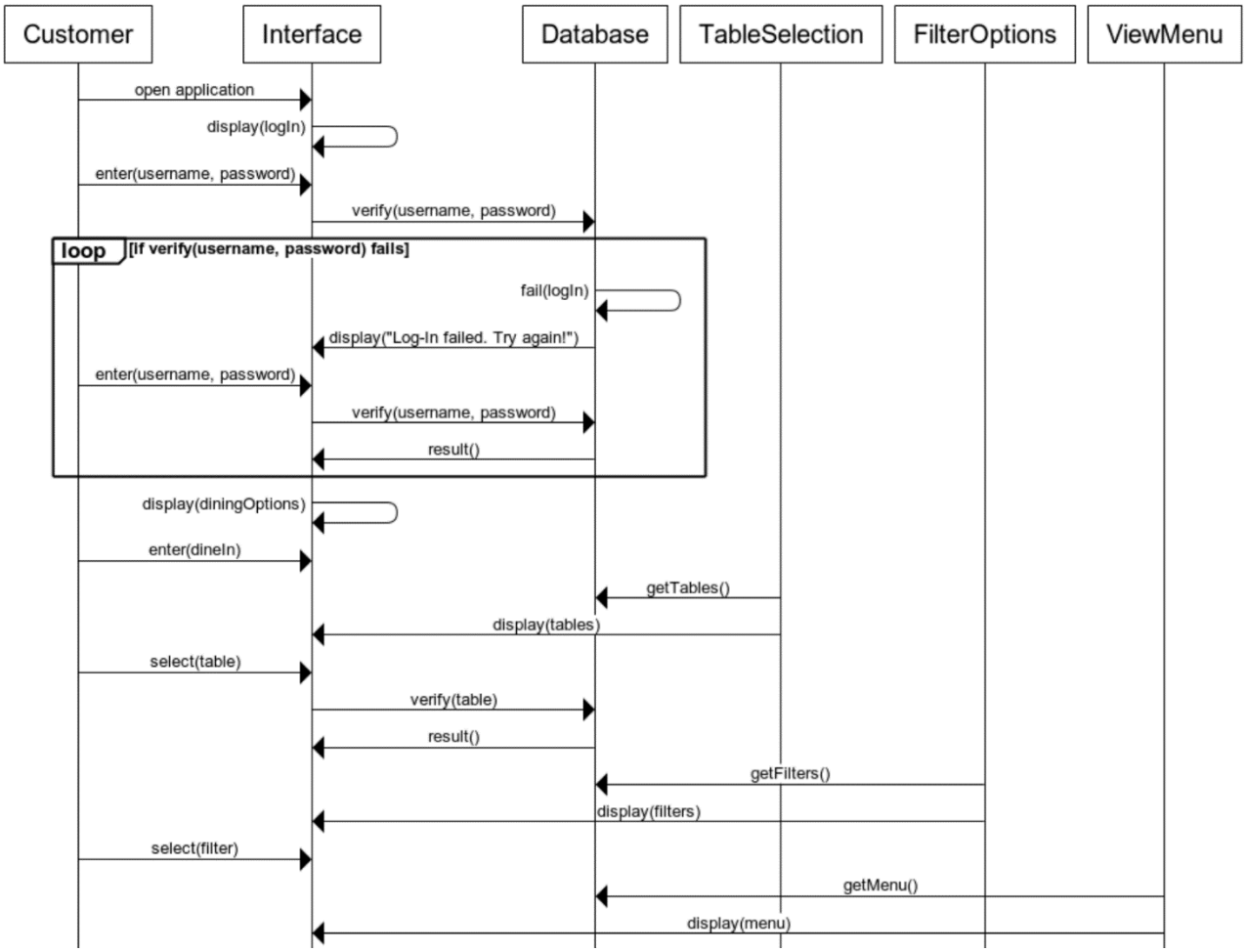


Figure 6: View Menu with Take-Out Selection



UC-7: Clocking-in/Clocking-out

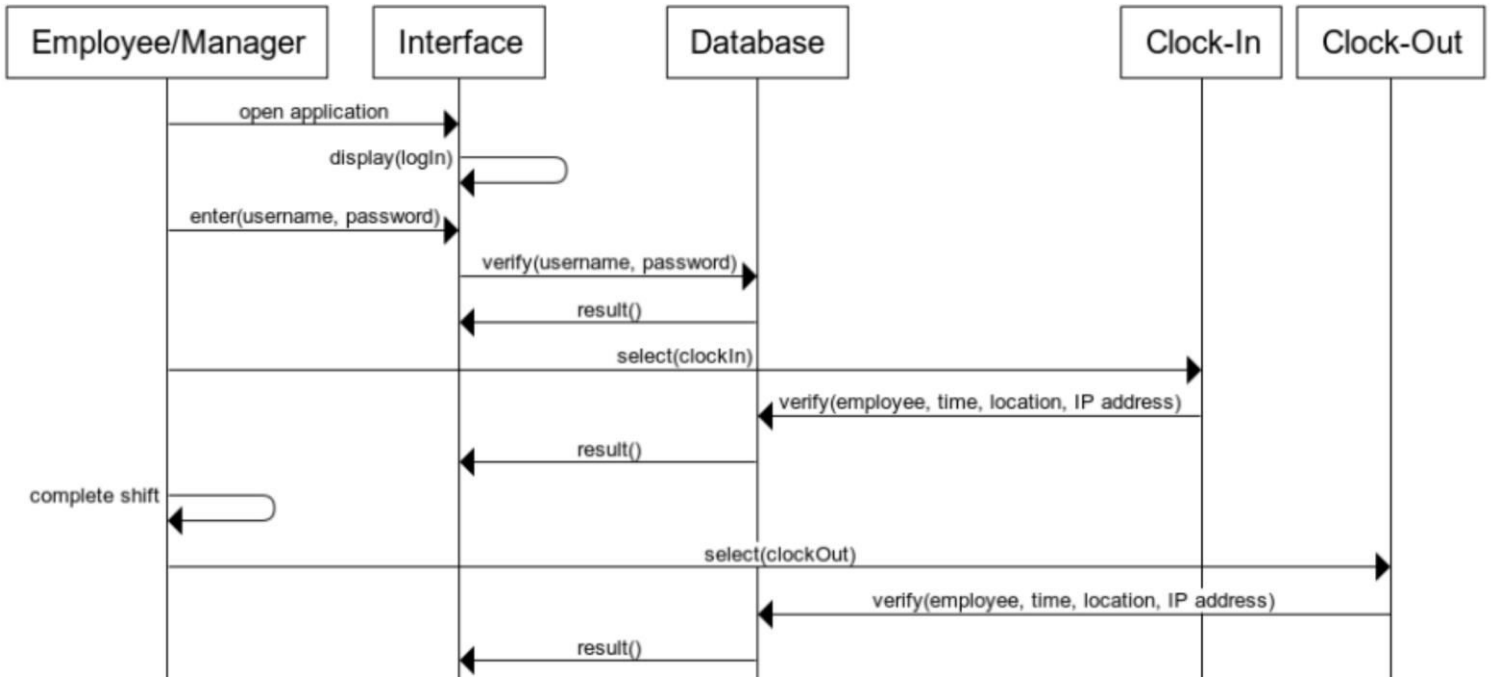
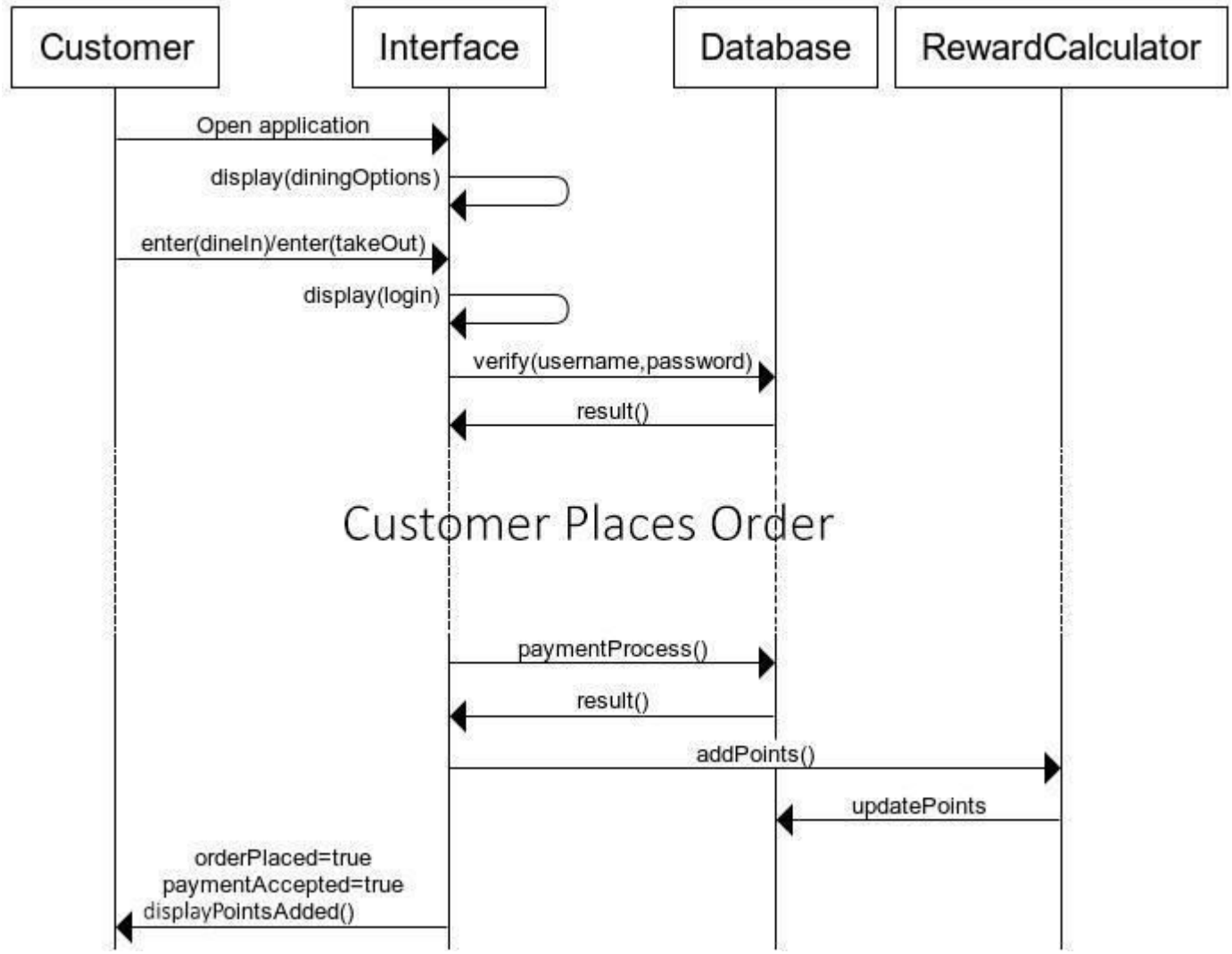


Figure 7: Clocking In/Clocking Out

The diagram above demonstrates the interaction between objects and/or actors in UC-7: Clocking In/Out. The user must interact with the interface in order to open the application and log in. Once the user enters their login details, the interface will check them with the database and confirm their credentials. The user is then brought to an interface where they are presented with the option to clock in. Once they select the option to clock in, the system logs the user's name, time of clock-in, location, and ip address. Later on in the day, the user is able to use the system again to clock out, where the system again logs the user's name, time, location, and IP address.

One design principle that is used is the High Cohesion Principle. All of the objects don't take on many computation responsibilities. Each object only focuses on the tasks they are specialized to do. The database has all the logged information required for objects to perform their task, while the objects simply have a calculation dedicated to their specified task (E.g. Clock in logs data at time of clock in, and vice versa for clock out).

UC-11: Earning Rewards

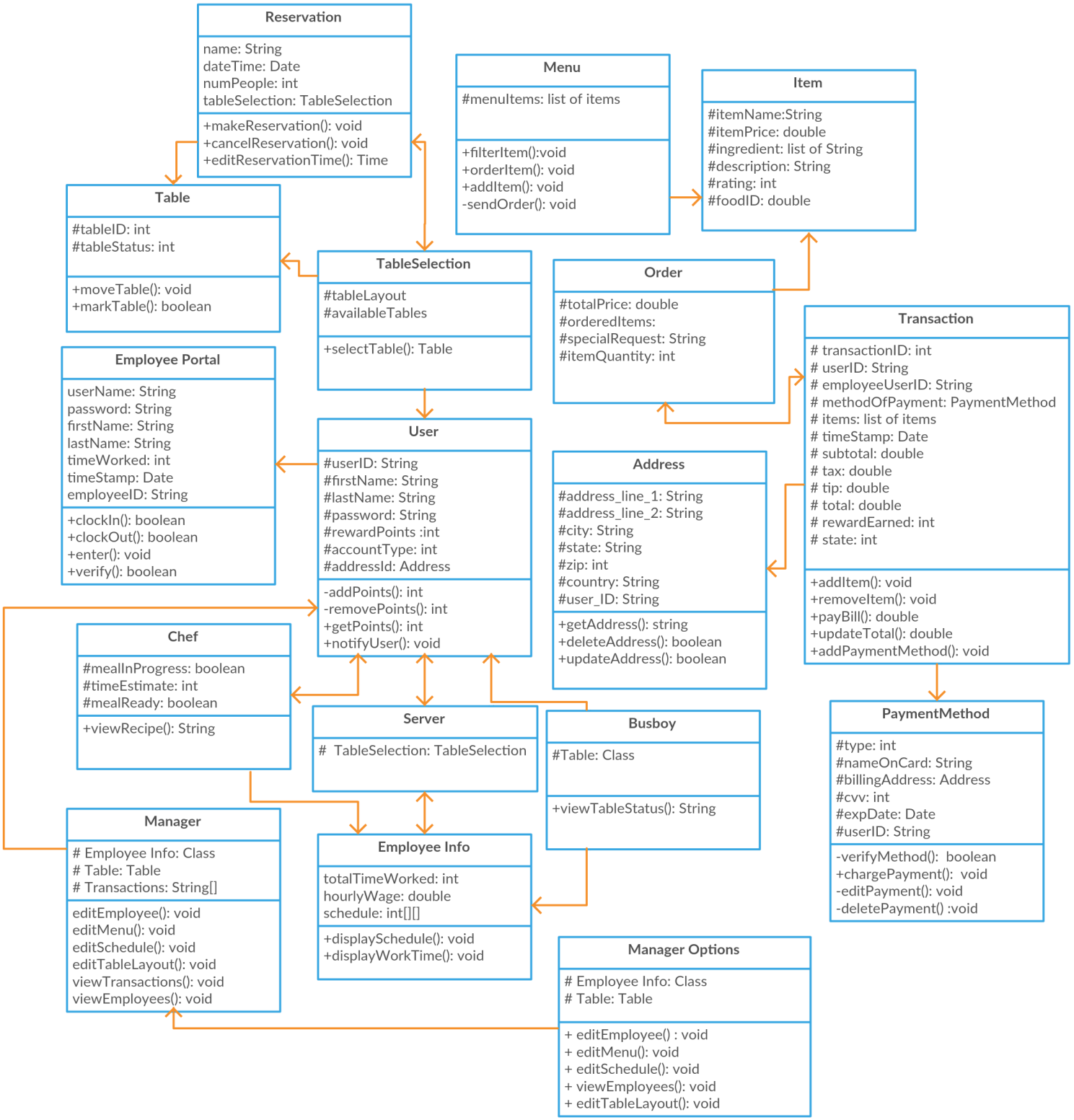


The diagram above demonstrates the interactions involved in UC-11: Earning Rewards. Initially the Customer will launch the application and select their dining option. The interface will prompt the user for their Once the user enters their login details, the interface will check them with the database and confirm their credentials. Upon sign in, the interface will display a menu and allow them to place their order (This process was left out of the diagram, as this process is lengthy, and doesn't differ in this use case). Once the customer is done with the ordering process, they will be directed to payment (see UC-2). Upon completing payment, the interface will carry details of the transaction to the reward calculator. The reward calculator will calculate an amount of points to give the user proportional to their total (as determined by the restaurant owner). The reward calculator will then update the database with the customer's new balance, then the interface will display their new balance to the user along with their receipt confirming the transaction was completed successfully.

One design principle that is used is the High Cohesion Principle. All of the objects don't have many computation responsibilities. The interface simply displays information to the user and prompts them for input. The database will store and return related information for each task. Also, the reward calculator will only be responsible for calculating rewards.

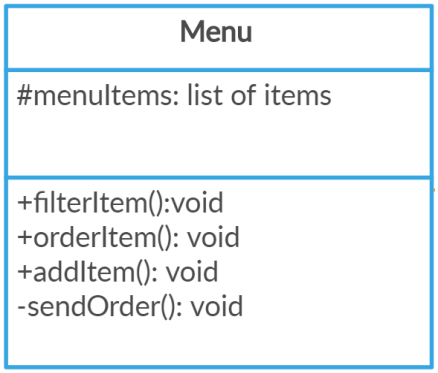
2. Class Diagrams and Interface Specification

2.1 Class Diagrams



2.2 Data Types and Operation Signatures

Class: Menu



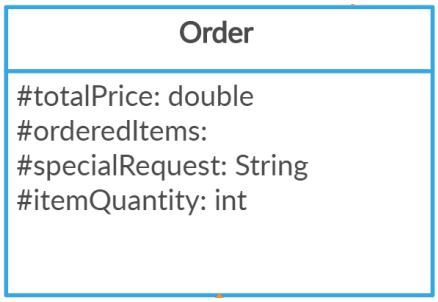
Attributes:

- menuItems: list of Item- food items available to the customer.

Methods:

- filterItem(preference)-allows allergens and preferences to be filtered out as indicated by the customer.
- removeItem(foodID#)- removes food from existing order- will take a food ID and quantity.
- addItem(foodID#)-adds food/beverage indicated by the customer to the current order.
- sendOrder(Order)- sends the order to the kitchen to be made.

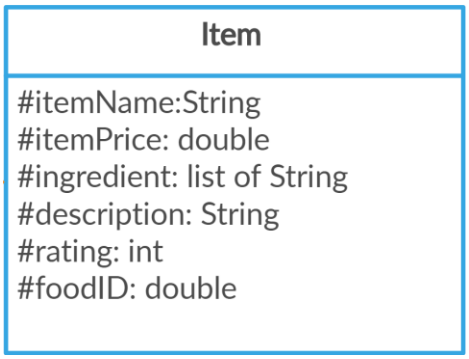
Class: Order



Attributes:

- totalPrice:double - the total price of all items in the order.
- orderedItems: Item[] - the list of food items in the cart picked by the customer.
- specialRequest: String - a place for customers to express any request not available in the UI.

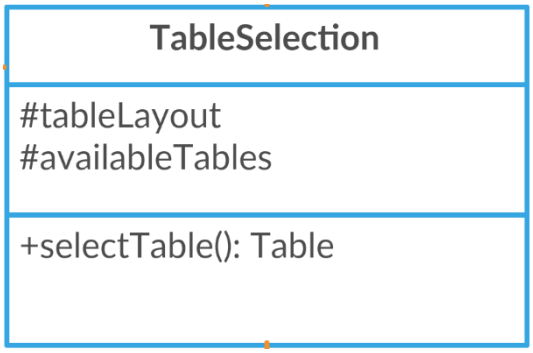
Class: Item



Attributes:

- itemName: String - The name of an item.
- itemPrice: double - price of each item.
- ingredients: list of String - a list of the different ingredients needed to make the item.
- description: String - each item is described with a few short sentences.
- rating: int - a number to indicate the overall satisfaction with an item calculated using previous user feedback.
- foodID: double- a unique number used to reference the item.
- itemQuantity: int - the number of a specific item ordered in that single order.

Class: TableSelection



Attributes:

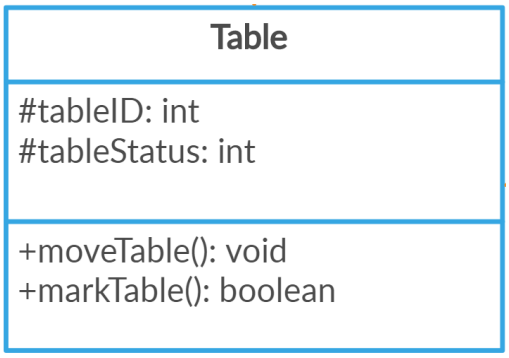
- tableLayout: two dimensional array of tables - Display the current table layout.

Methods:

- selectTable() - Customer has to select a table.
- verifyTable() - This is will verify that the table that the customer selected is still open (to make sure they do not select an occupied table).

- displayTables() - Customer is shown a picture of the available tables, with unavailable table shown greyed out.

Class: Table



This class is made so that status of tables can be altered and table layout can be changed.

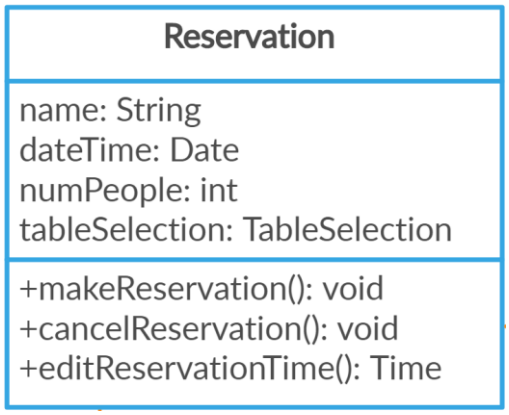
Attributes:

- tableId: int - Each table is assigned a specific number, so that they are easy to keep track of.
- tableStatus: int - Returns a 0 for when a table is unoccupied/open and returns a 1 for when the table is occupied and returns a 2 for when a table is dirty and unoccupied, so that a busboy can know when to clean a table

Methods:

- moveTable(): void - A manager is allowed to move and rearrange the tables in the layout.
- markTable(): boolean - A table will be able to be marked, so that it's status can be updated.

Class: Reservation



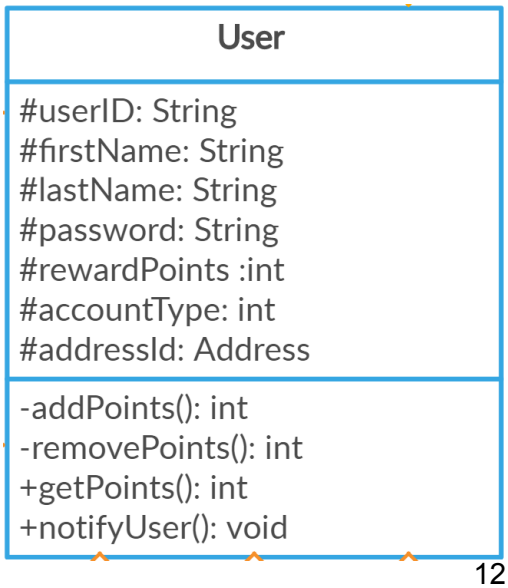
Attributes:

- name: String - The name of the reservation group.
- Date: Date - The date of the transaction.
- numPeople: int - The amount of people for the reservation.
- TableSelection: Class - Inherits the TableSelection Class in order for customers to have the option to select their table.

Methods:

- makeReservation() - Customers can make reservations ahead of time.
- cancelReservation() - Customers can cancel the reservations they made.
- editReservationTime() - Customers can edit their reservation time.

Class: User



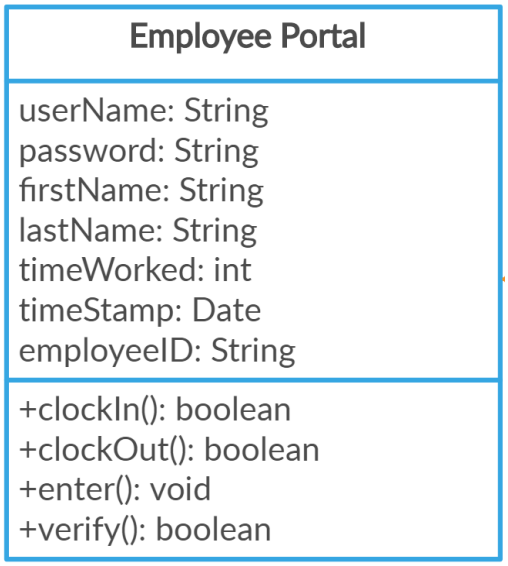
Attributes:

- userId: String - The ID of the user.
- firstName: String - The first name of the user.
- lastName: String- The last name of the user.
- userName: String - The username for the user.
- password: String - The password for the user's account.
- rewardPoints: int - How many rewards points the user has.
- accountType: int - The type of user account.
- address: String - The user's address.

Methods:

- addPoints() - Add an integer of reward points to the user's balance.
- removePoints() - Remove an integer of reward points to the user's balance.
- getPoints() - Get the user's point balance.
- notifyUser() - If the user has an active session, send them a message.

Class: Employee Portal



This class is made to allow employees to clock-in and clock-out of their shifts (including breaks).

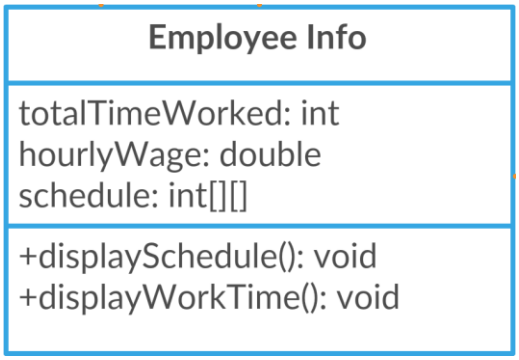
Attributes:

- userName: String - This is the individual employee's self-made username to be able to have an account.
- password: String - This is the individual employee's self-made password to keep their profile protected.
- firstName: String - This is the individual employee's first name.
- lastName: String - This is the individual employee's last name.
- timeWorked: int - This will keep track of the amount of time that the individual employee has worked during the current shift.
- timeStamp: Date - The time and date of the clock-in/clock-out.
- employeeID: String - This is an employee specific ID, so that the system and manager can identify the employee's role and have the ability to edit and track.

Methods:

- clockIn(): boolean - Employee has the ability to clock in when beginning their shift.
- clockOut(): boolean - Employee has the ability to clock out when ending their shift.
- enter(username, password) - Employee has to enter their username and password to login at the beginning of the shift.
- verify(username, password) - This will check if the username and password entered are correct. If not, the employee has to enter again.

Class: Employee Info



This class is made to hold employee information, if the employee wishes to access it.

Attributes:

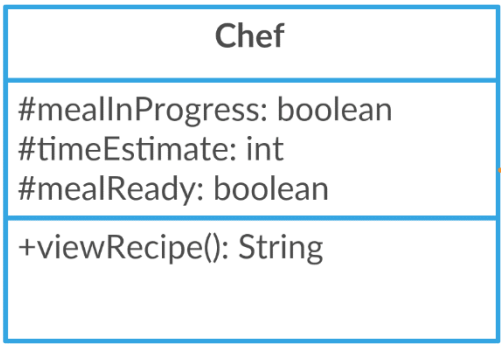
- totalTimeWorked: int - This will keep track of the total amount of time that the individual employee has worked during the payment cycle.
- hourlyWage: double - This will keep track of the individual employee's working hourly wage.
- schedule: int[][] - This will keep track of the work schedule during the current pay cycle.

Methods:

- displaySchedule(): int[][] - This will display a 2D array that holds the scheduling information.
- displayWorkTime(): int - This will display the employee's total amount of time worked during the pay cycle.

Class: Chef

This class is made to display the Chef Interface and the options for the chef.



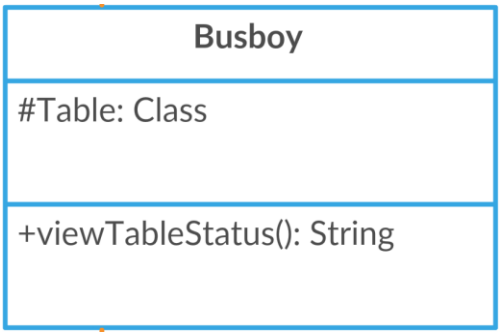
Attributes:

- mealInProgress: boolean - This is for the chef to track which meals they have began to make, and helps with giving a time estimate to the customer.
- timeEstimate: int - This is the estimate for how long the customer's meal might take to arrive.
- mealReady: boolean - This is to notify other users when the meal is ready.

Methods:

- viewRecipe(): String - This will display the recipe if the chef desires - in case they need a reminder.

Class: Busboy



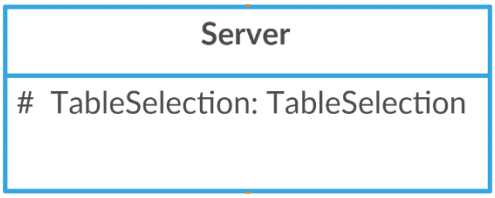
Attributes:

- Table: Class - Inherits the Table class in order to be able to see marked tables and mark them.

Methods:

- viewTableStatus(): int - This will allow the busboy to see the statuses of the tables.

Class: Server

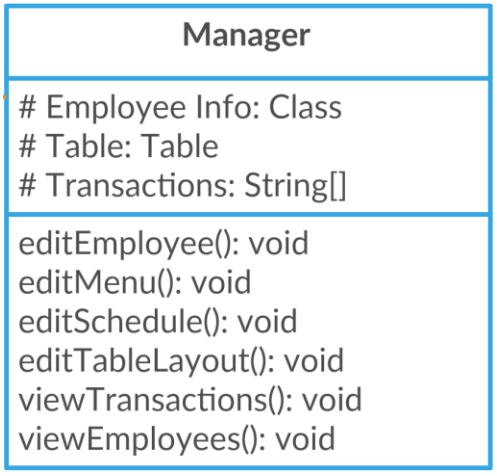


This class is made to display the Waiter/Waitress Interface and the options for the waiter/waitress.

Attributes:

- TableSelection: Class - Inherits the TableSelection class, so that the waiter/waitress can select a table for the customer if he hasn't done so yet.

Class: Manager



This class is made to display the options specifically for a manager.

Attributes:

- Employee Info: Class - Inherits Employee Info class.
- Table: Class - Inherits Table class.
- Transactions: String[] - Transactions that were made.

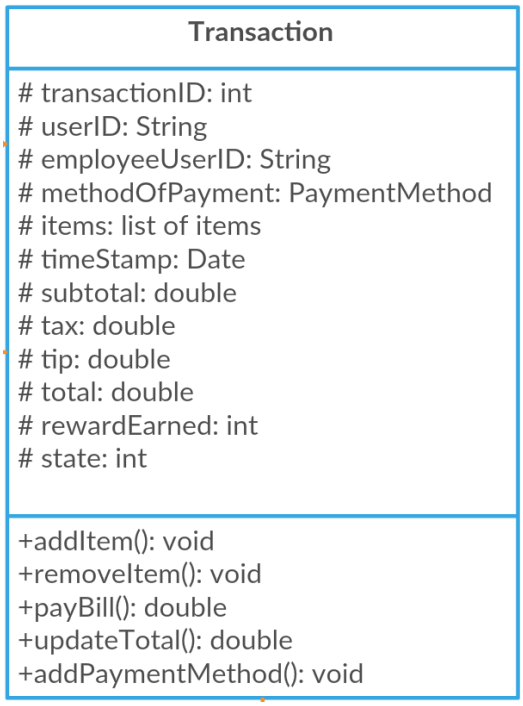
Methods:

- editMenu(): void - This allows the manager to edit the available menu.
- editEmployee(employeeID): void - This allows the manager to edit the information of an individual employee.
- editSchedule(): void - This allows the manager to edit the schedule for the current pay cycle.
- viewEmployees(): void - This allows the manager to view all employees currently working.

- editTableLayout(): void - This allows the manager to edit the table layout of the restaurant.
- viewTransactions(): void - This allows the manager to view all recent transactions.

Class: Transaction

This class is created to help maintain the information that is associated with a transaction occurring.



Attributes:

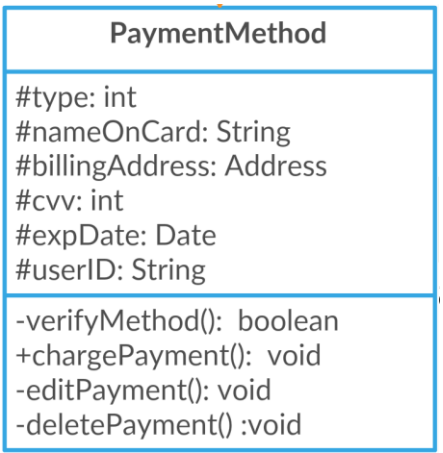
- transactionId: int- This will keep track of the designated ID of the transaction, which will start counting up with the first item having an ID of 1.
- userId: String - This will hold the ID of the user making the transaction.
- employeeUserId: String - The user ID of the employee.
- methodOfPayment: PaymentMethod - keep track of how the payment was made (card, cash, etc) as well as the information that is associated with paying by card such as card number, CVV code, and expiration date.
- items: list of Items - This attribute will be a list of all of the items associated with the order that the customer must pay for.
- timeStamp: Date - The time and date that the transaction was enacted.
- subtotal: double - The subtotal amount of the order.
- tax: double - The tax amount on the order.
- tip: double - The tip amount that is being paid.
- total: double - The amount of the order total.
- rewardsEarned: int - The amount of rewards earned.
- state : int - This is the status of the transaction, 0 pending, -1 indicates an error, 1 is a success.

Methods:

- addItem() - boolean - Adds and item to the transaction.
- removeItem() - boolean - Removes an item from the transaction.
- payBill() - void - Allows user to pay their bill.
- updateTotal() - double - Updates the total of the transaction.
- addPaymentMethod() - boolean - Allows user to add a payment method.

Class: PaymentMethod

This class will keep track of all of the details about the method of payment, if it was done by cash or card, and if the transaction is completed by card then the card information will be stored.



Attributes:

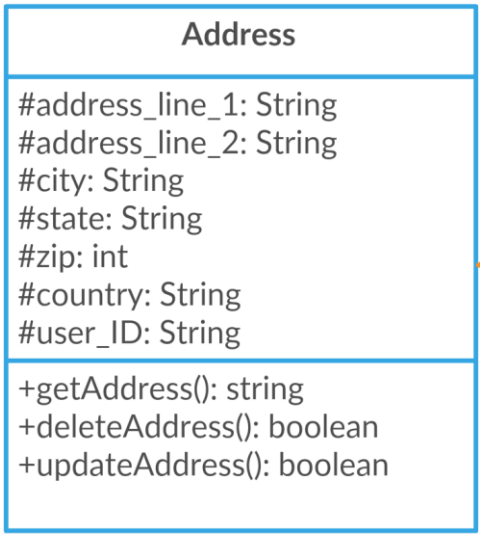
- type: int - 0 for Cash, 1 for Credit Card, 2 for Debit Card - The different potential methods of payment.
- nameOnCard: string - Stores name on Debit/Credit Card - The name associated with the card that is paying.
- billingAddress: Object of Type Address - The billing address associated with the card that is paying.
- cvv: int - stores CVV of Credit/Debit Card - The cvv associated with the card that is paying.
- expDate: Date- the expiration date that is associated with the card that is paying.
- userId: - the Customer ID that is associated with the account that is paying.

Method:

- verifyMethod() - boolean - Checks if it is a valid method of payment.
- chargePayment() - void - Allows user to change their payment.
- editPayment() - void - Allows user to edit their payment.
- deletePayment() - void - Allows user to delete their payment.

Class: Address

This class stores the strings/ints required for address.



Attributes:

- address_line_1: String - Stores address line 1.
- address_line_2: String - Stores address line 2.
- city: String - Stores the name of the city.
- state: String - Stores the name of the state.
- zip: int - Stores the ZIP code.
- country: String - Stores the name of the country.
- user_id: String - The user ID corresponding to the address.

Method:

- getAddress() - String - Gets the address of a user.
- deleteAddress() - boolean - Allows user to delete their address.
- updateAddress() - boolean - Allows user to update their address.

2.3 Traceability Matrix

Domain Concepts	Menu	Order	Item	TableSelection	Table	Reservation	User	EmployeePortal	EmployeeInfo	Chef	Busboy	Server	Manager	Transaction	PaymentMethod	Address
Customer Profile		X					X							X	X	X
Interface	X	X	X	X	X	X	X	X		X	X	X	X		X	
Payment System							X							X	X	
Food Status		X								X		X				
Order Queue		X								X						
Controller	X			X		X								X	X	
Communicator		X					X			X	X	X	X			
Floorplan				X	X							X	X			
Employee Profile							X	X	X	X	X	X	X			X
Table Status				X	X	X					X	X	X			
Reward System							X							X		
DB Connection		X	X		X	X	X		X					X	X	X

- Customer Profile:
  - User: Allows customer to view and edit account specific data.
  - PaymentMethod: Allows customer to select preferred payment method.
  - Address: Allows customer to store their personal address
- Interface:
  - Menu: The restaurant's menu is accessible via the interface.
  - Order: Orders can be placed through the interface.
  - Item: All available items are presented on the interface.
  - TableSelection: Table selection is done via the interface.
  - Table: Table availability and status is available via the interface.
  - Reservation: Table reservation is done via the app interface.
  - User: User data is presented via the app interface.
  - Employee Portal: The portal is accessible via the app interface.
  - Chef: Can access their respective UI via the interface.
  - Busboy: Can access their respective UI via the interface.
  - Server: Can access their respective UI via the interface.
  - Manager: Can access their respective UI via the interface.

- PaymentMethod: Chosen through the interface.

- Payment System:
  - User: The purchase is logged to the user's account after the payment is made.
  - Transaction: The system attempts to validate and confirm payment has been made.
  - Payment Method: The system uses the selected payment method in order to complete payment.
- Food Status:
  - Order: Food Status initiates once an order is placed.
  - Chef: Chef is able to update Food Status depending on the stage of the order.
  - Server: Server is able to see once status is complete and clear it.
- Order Queue:
  - Order: Ordering a food item adds it to the queue.
  - Chef: Chef is able to view Order Queue and dequeue them as they handle it.
- Controller:
  - Menu: Controller requires that items be chosen from the menu.
  - TableSelection: Controller requires that a valid table is selected.
  - Reservation: Controller requires that the customer enters reservation details.
  - PaymentMethod: Controller requires a valid Payment Method from customer.
- Communicator:
  - Order: Communicator allows the customer's order to be communicated to the chef via queue.
  - User: Allows accounts to validate information from server, such as login.
  - Chef: Allows Chefs to receive important notifications and data.
  - Busboy: Allows Busboys to receive important notifications and table status updates.
  - Server: Allows Servers to receive important notifications and table status updates.
  - Manager: Allows Managers to receive important notifications and data.
- Floorplan:
  - TableSelection: Allows the table to be moved to be selected.
  - Table: Floorplan will edit the properties of the Table after it is changed.
  - Server: Servers will be able to edit the current floorplan.
  - Manager: Managers will be able to edit the current floorplan.
- Employee Profile:
  - User: Contains employee personal information.
  - Employee Portal: Allows employees to clock in/out and view work status.
  - Employee Info: Allows employees to view their current earnings and status.
  - Chef: Chefs are able to see all relevant personal information via their portal.
  - Busboy: Busboys are able to see all relevant personal information via their portal.
  - Server: Servers are able to see all relevant personal information via their portal.
  - Manager: Managers are able to see all relevant personal information via their portal.
  - Address: Employee home addresses are stored on their profile.
- Table Status:
  - Table Selection: Allows users to select a table and view its status.
  - Table: Contains the different variables pertaining to table status.

- Reservation: Allows users to preserve table status for a later time.
- Busboy: Can be viewed and edited by a Busboy after it is cleaned.
- Server: Can be viewed and edited by a Server after it is being used.
- Manager: Can be viewed and edited by a Manager depending on restaurant status.

- Reward System:
  - User: All rewards points are saved to a user's profile.
  - Transaction: Points allocated after a transaction, can be used before a transaction.
- DB Connection:
  - Order: Orders will be sent to the database once sent by the customer.
  - Table: Table information is stored on the database.
  - Reservation: Reservation information will be stored on the database.
  - User: User account information will be stored on the database.
  - Employee Info: Employee information will be stored on the database.
  - Transaction: Transaction history will be stored on the database.
  - Payment Method: Saved and preferred payment methods will be stored on the database.
  - Address: Saved address will be stored on the database.

3 System Architecture and System Design

3.1 Architectural Styles

For our application, we are using the Layered Pattern mode which is commonly used for general desktop applications. The three layers are presentation, application, and data layers. The presentation layer is seen by the users on the front-end, both customers and restaurant staff. Application and data layers are on the back-end and keep track of user-entered information, as well as respond to commands from the presentation layer. This model is appropriate for our application as the layers are abstracted from each other to therefore run parallel with the code.

Our application also uses the Client-Server architecture model which is a one to many relationships. The server provides/stores data. In the restaurant, the servers are the many different interfaces for customers and for the restaurant staff. Users are the clients that have to enter information into three interfaces for; ordering food from the menu, making reservations, logging in to the rewards system, and making payments. The employees are clients who use the clocking in/out server to log their shifts so they will be paid. Managers are clients who use their options server to edit employee information and scheduling, edit the menu, and edit table information.

Another architectural style utilized is the Peer-to-Peer model. This model is a distributed application architecture that partitions tasks between peers. This will be used for the payment of the order by a customer to the restaurant and the model is decentralized so every payment gets processed separately. In our application this model is used when if the customer pays by card then the peer to peer model becomes a payment service to transfer funds from customers bank account to the restaurant account.

3.2 Identifying Subsystems

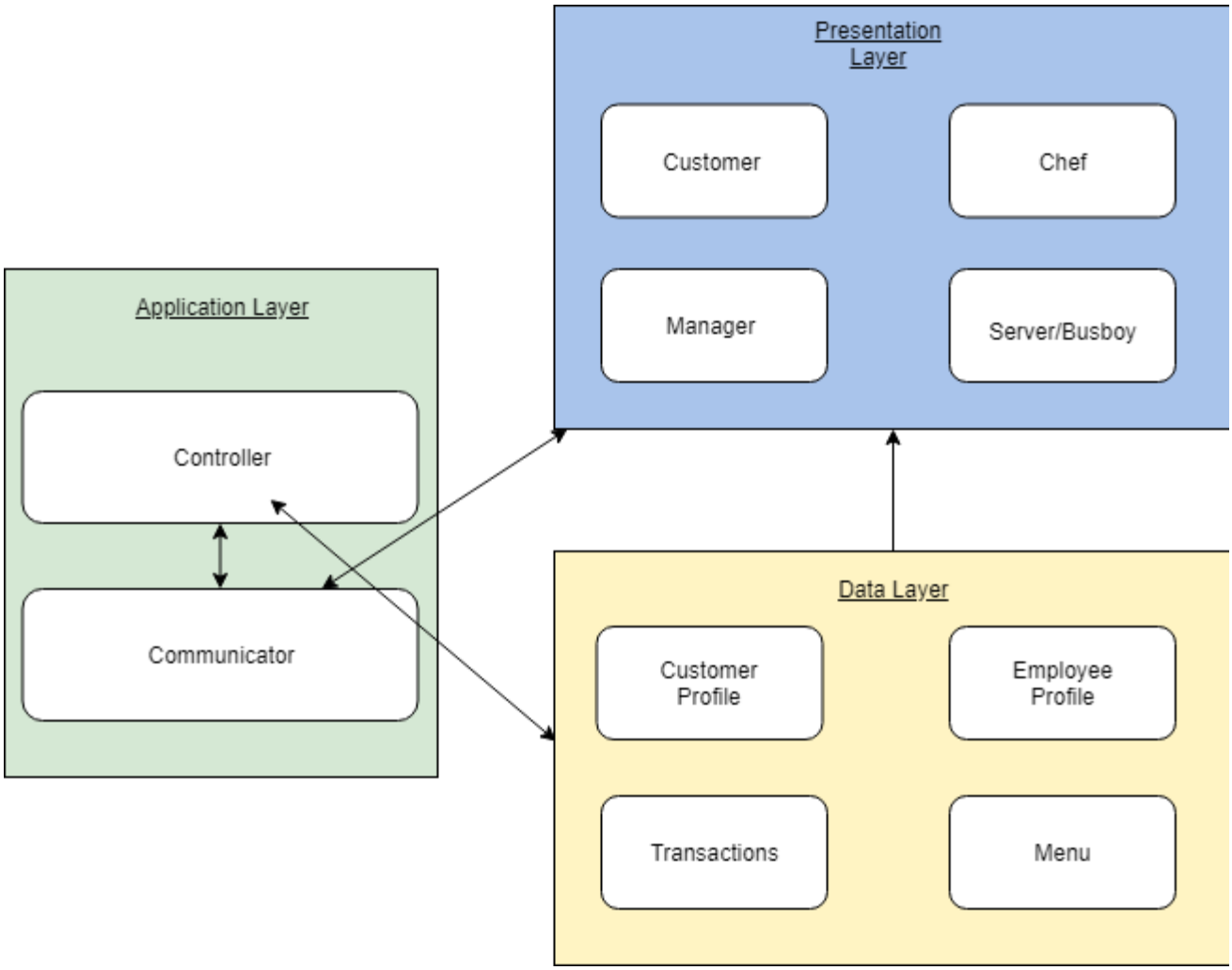


Figure 9: UML Package Diagram

The subsystem above displays our three-layer architecture - which consists of a Presentation Layer, an Application Layer, and a Data Layer. Each layer has a specific responsibility to the entire system.

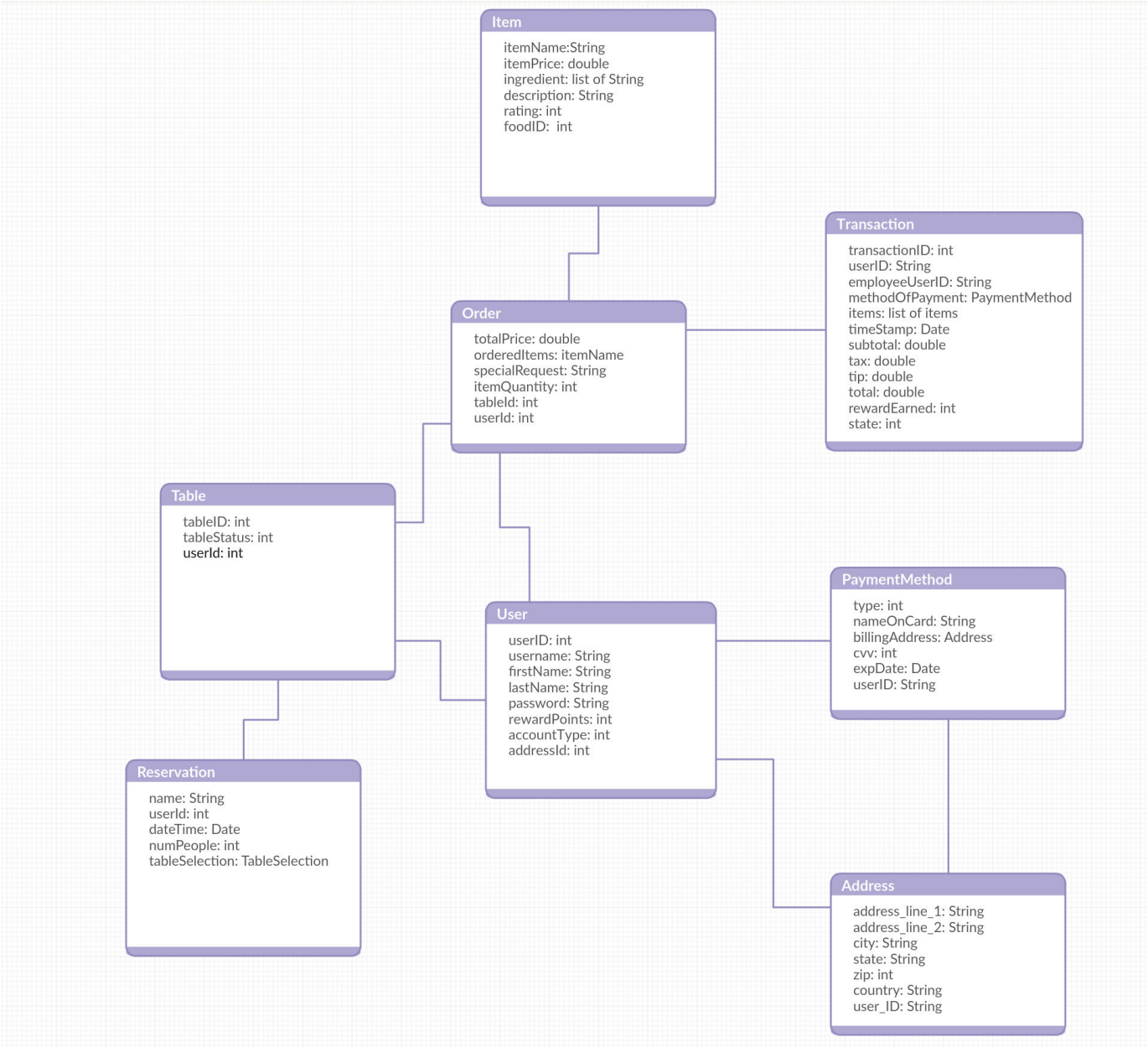
The presentation layer handles the user interface level. This layer holds the information dealing with the Bartender/Chef, Manager, Busboy/Waiter/Waitress, and Customer Interfaces. The application layer is the middle tier of this architecture, which handles communication between the top and bottom layers, essentially managing the overall operation. It will run the business logic, which is the set of rules required for running the application for the guidelines given by the organization. Contained within this layer is the Communicator and Controller. The Controller is utilized to facilitate tasks between the layers and the Communicator is used to pass information between the layers. The lowest layer is the data layer, which deals with the storage and retrieval of data. This tier will hold information pertaining to the Employee and Customer Profiles, as well as the available menu and restaurant transactions.

Each layer doesn't need to worry about the other layers, due to separation of concerns. Also, due to layers of isolation, changes made in one layer don't generally affect the other layers. Because of this, the architecture makes testing and diagnosing issues more manageable.

3.3 Mapping Subsystems to Hardware

For our software there will be a client running on tablets and cellphones in the restaurant. There will also be a database server that the client will communicate with over Wi-Fi. The application can be ran on many different devices as long as it meets the requirements listed in Section 3.7 Hardware Requirements. React Native will be used to run our client on various mobile devices. Our system will have a RESTful API and will communicate with our servers in the fetch() phase. POST and GET are requests that will come from the server and communicate with the client. POST requests supply additional data from the client to the server. GET requests include all required data in the URL. There also is a physical server that we will control using Node.js. For the server Node.js contains packages Express and Sequelize which helps the communication between the server and the database. Express is a minimal and flexible Node.js web application framework that provides a robust set of features to develop web and mobile applications. Express is used to create handle routing and process requests from the client. It facilitates the rapid development of Node based web applications. Sequelize is a promise-based ORM for Node.js, which will enable the object to create objects from our relational database. This supports MySQL and features solid transaction support, relations, read replication and more. For our database we will be using MySQL. MySQL is a driver for Node JS. MySQL is a Relational Database Management System that uses Structured Query Language (SQL). SQL helps with adding, accessing and managing content in a database. The data will communicate with the server with the help of SQL.

3.4 Persistent Data Storage



Our software does need to store persistent data that needs to outlive a single execution of the system. For one, the Customer Profile needs to be saved. Within the Customer Profile, our software needs to remember the customer's first and last name, username and password, as well as their Reward Points earned and payment information. The Employee Profile also needs to be saved. Within the Employee Profile, our software needs to remember the employee's first and last name, username and password, salary, time worked, work schedule, etc. Our software will also need to hold onto the restaurant's transactions, so that the manager will be able to access it whenever they please and have the ability to track earnings (total revenue and profit). Since the manager should also be able to adjust the table layout, our software needs to have the ability to remember the restaurant's current table layout. Other important information that needs to be saved is the available menu, along with the totaled ratings calculated for each item, item prices, item names, the recipes, and item number. This information will all be stored in a MySQL database running on our web server.

3.5 Network Protocol

In order to complete the restaurant automation application, our team is implementing a REST API. A REST API will perform actions on the server based on HTTP POST and GET requests. HTTP REST lends itself well to a client server architecture. The front-end of the application will be built with React Native, which will handle performing the requests. The user interacts with the application and when they perform a task, a request will be sent to a server via HTTP where the information will be processed. For example, if the user orders food after clicking confirm, the order will be sent via HTTP POST request to the server which will respond back with information including estimated wait time, which is necessary for the client to display to user. Another example is when the user seeks information about how many reward points the user earned, a GET request from the client to the server will be issued requesting the user's reward points balance, the server will process this request and, in its response, body will include the balance.

3.6 Global Control Flow

Execution Orderness: This is procedure driven in a "linear" fashion. The order of the application differs depending on the different users. The initial action is the same for all users, which is logging in or continuing as a guest. If the user is a customer, then the order of the application are as follows: user will be able to choose dining option, select a table, filter out allergies/dietary restrictions, select items from menu into cart, then proceed to cart, with an option to view the order progress, and finally to the payment screen. The user has very little control of the execution orderness. The owner/manager experience is less linear than the customer. They have the option to view the table layout and change it whenever needed. The manager can view/manage employees/employee information through an employee management screen. The employees also have more flexibility in their application order, they can few the table selection and move the table around, clock in and clock out, and view orders.

Time Dependency: While there are no timers in the application, some things do rely on the physical time such as keeping track of orders. The customer/user will be able to view the order progress. This will depend on when the chef finishes preparing the order and notifying when done. Also, the employees will be having a clocking in/out view where the timer will keep track of the time from when the employee signs in to the time the user signs out this will have a time stamp to each employeeID.

Concurrency: We utilize different threads in our database because you can do multithreading as every order will be taken care separately. Hence, there are some threads running the UI and another running the backend.

3.7 Hardware Requirements

Our software application will be able to be run on mobile devices such as smartphones and tablets utilizing network support (WiFi or Mobile data). This software also needs GPS usage. This software will be compatible with iOS and Android operating systems. For iOS, we are going to support the latest operating system, iOS 12, which will cover 80% of the iPhones in circulation. For the Android operating system, we are aiming to support API 23-Marshmallow, by supporting this operating system we will be supporting 71% of current Android phones. This software will be touchscreen interactive. Our software will have a color display, with a minimum resolution of 1330 x 700 pixels upto 2960 x 1440 pixels. There needs to be a minimum of 50 Megabytes of storage. The application will communicate over the restaurant Wi-Fi and will push and pull requests to communicate with a server.

4 Algorithms and Data Structure

```
const Sequelize = require('sequelize');
const sequelize = require('./sequelizeConf.js');

module.exports = (sequelize, DataTypes)=>
{
   return sequelize.define('item',
   {
      itemName: Sequelize.STRING,
      itemPrice: Sequalize.DOUBLE,
      ingredient: Sequalize.STRING,
      description: Sequalize.STRING,
      rating: Sequalize.INTEGER,
      foodId: Sequalize.DOUBLE
   })
}
item.belongsToMany(ingredient);
ingredient.belongsToMany(item);es
```

4.1 Algorithms

For employee working information and clocking in/out, Team C's algorithms will be calculations for how much an employee works. One calculation will be put in a method called hoursWorked(). This method will take the timeOut, and timeIn variables, and convert both to "military time" (24-hour time). This will make calculations simple. Next,

the method will subtract timeIn from timeOut, and return the result. The Salary() method will calculate the annual salary of the given worker given hourly wage (wage), and number of hours worked (hoursWorked). This method gives a running total of a worker's salary for the year. The worker's salary for the day is calculated by hoursWorked * wage, this salary is added to expectedPay, then this process is repeated for every working day throughout the year.

When a customer is making a transaction, the order which they are making will be converted into a transaction object. When the transaction is created, a subtotal is calculated by adding the price of all items in the order, then a tax rate is applied to calculate a total. Once this information is presented to the user, and they enter their tip, that is added to calculate the total amount due for the transaction. The stripe API will be used to implement payment, and so all the information attached to this transaction object will be used to call the stripe API. Once the transaction goes through, the rewards system will increment a counter of how many visits the visiting customer has. Once they reach a set amount of visits (decided by the restaurant), they will earn a reward, which will be a set amount off of their next purchase (as decided by the restaurant as well)

For the menu side of our application, which will be worked on by Team A, customers will have the option to view a menu. Once customers view the menu, they will have the option to browse through everything in the menu or they can filter things out. For example, if a customer has a food allergy they can chose to filter out all items that have that ingredient in the meal. This is very helpful for customers with food allergies and also other dietary restrictions. Once we an ingredient is selected on the filter, out system will check the database for all the items including that ingredient. Then our database will send back the list of foods without the filtered ingredient and only display food items without the filtered ingredient. Once customers know what they want, they will be able to add and remove food items from their order as they please.

4.2 Data Structures

We can utilize is queue data structure for the Chef Interface. A queue is an ordered collection of items where the addition of new items happens at one end and the removal of existing items occur at the other end. This structure of a queue using the ordering principle of first in first out also known as "first come first served". In the Chef Interface, the queue is used when orders come in from the system the chef is notified first of that one and works on that meal and then serves it out which removes it from the queue. This performance of accessing is O(n) and to remove is O(1) this is a very efficient data structures to use for this scenario.

There will be a menu table within the SQL database. A database is good for querying and selecting items based on attributes. This tables primary key will be foodId and the other columns necessary will be food name, food price, food ingredients and to improve efficiency it may be helpful to add common allergies and restrictions as columns for example, contains nuts in a meal.

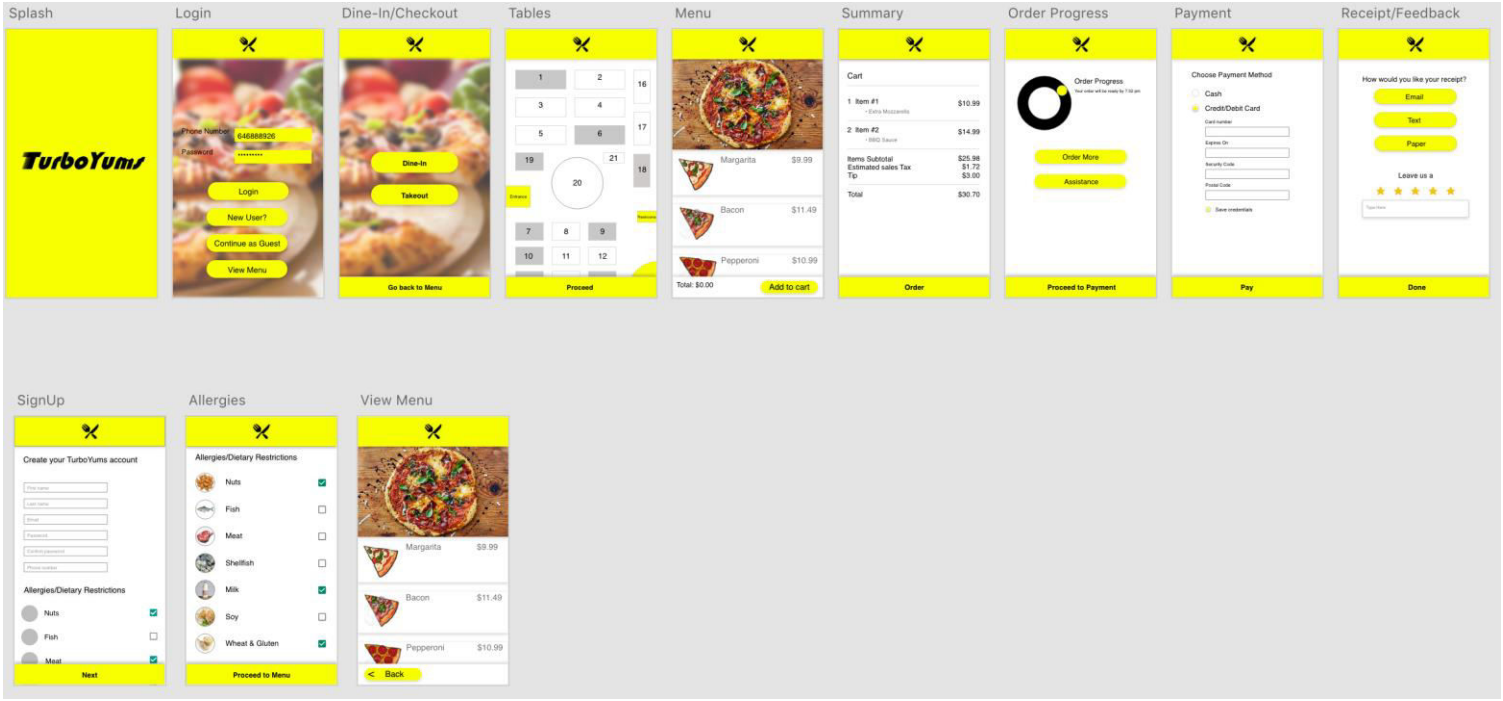
In order to keep track of employee working information, such as time worked and current status, we will use a SQL database. The table employee in the database will have primary key, employee's username, so that we can easily find an entry pertaining to that employee. The columns keep track of the employee's time clocked in, time clocked

out, whether or not the clocking was done on site, their current clock status, the total numbers of hours worked, each employee's individual wage, and the estimated pay that is to be received on the next pay cycle.

When a customer is preparing an order, the system will create an order object, which will include a collection of items included in the order, as well as an associated table, customer, and server. When the customer goes to complete a purchase, this order object will be used to create a transaction object, which will include method of payment information, as well as a total. The method of payment will be represented by an object, there will be a field dictating what method of payment this is (cash, card, etc..), a field associating it with a user, and also fields detailing the information regarding the method of payment (card number, expiration date, billing address…). When a payment goes through, This will also update the User object's rewards information. All the objects used are stored tables in our relational SQL database.

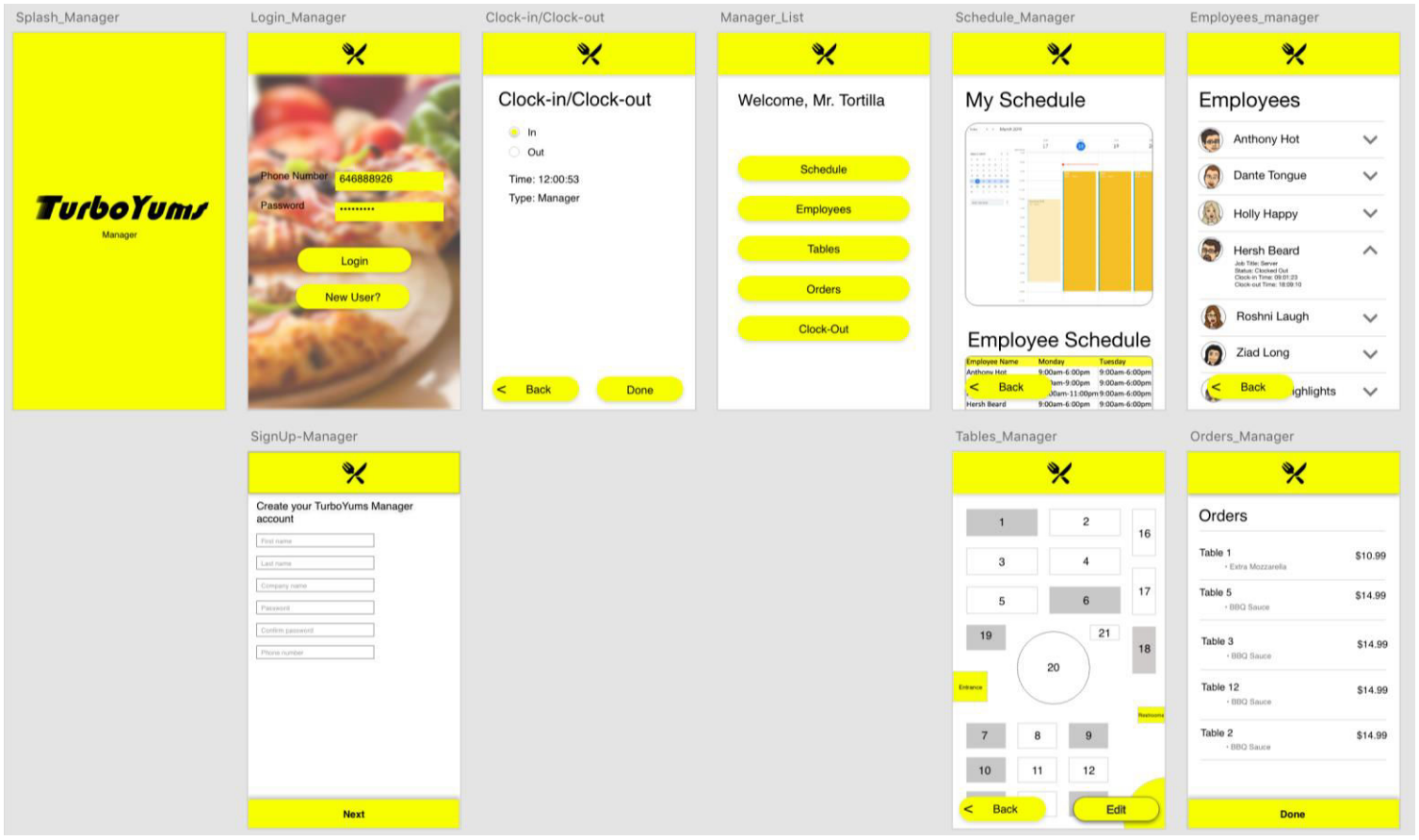
5 User Interface Design and Implementation

Customer Interface



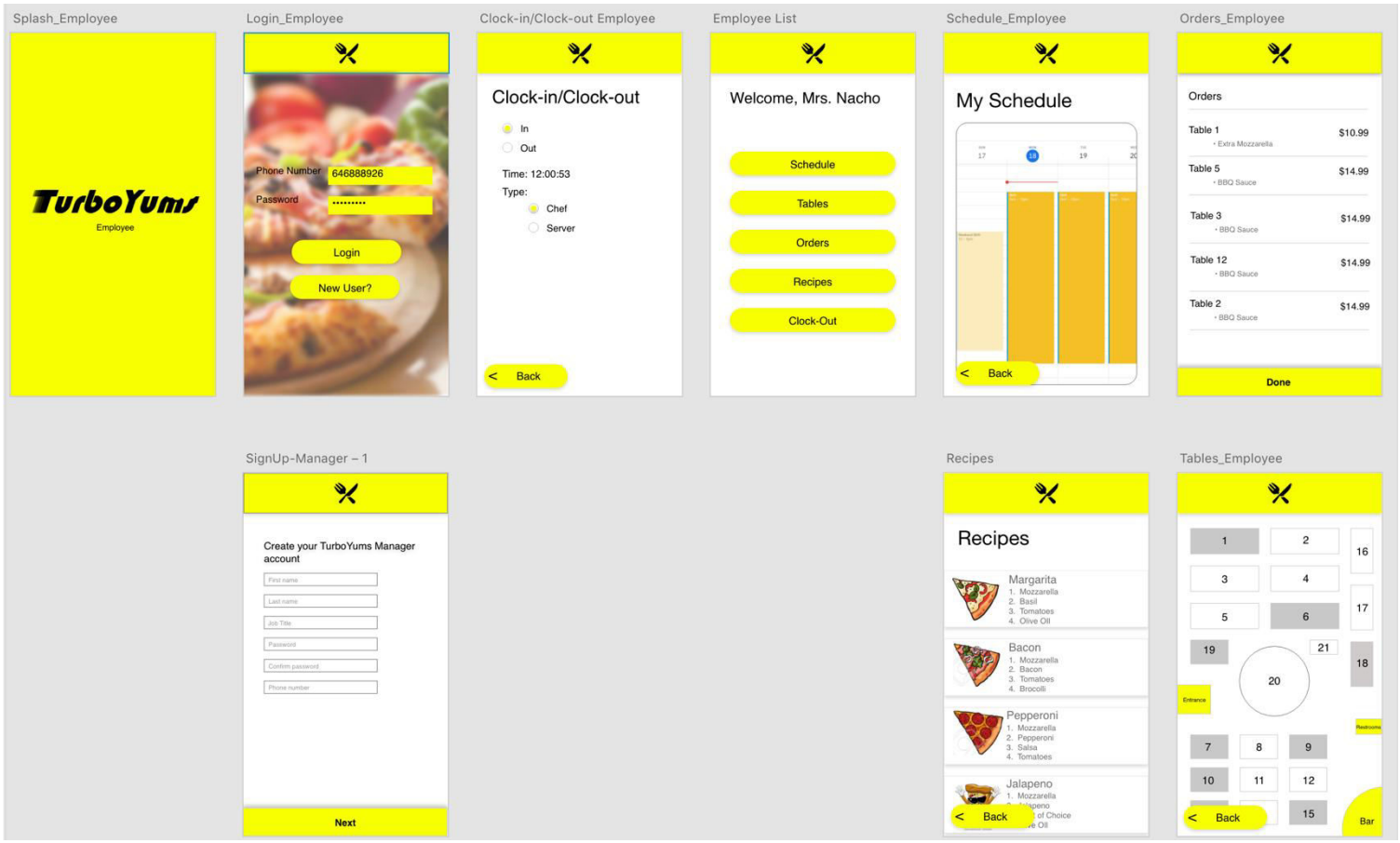
The customer interface is pretty straight forward, allowing the user to login/signup/continue as guest/view menu right off the home page. Clicking on the different options takes the user through different Interface flows as shown in the diagram. For example, if a returning customer wants to login and order food after choosing a table and order again before checking out using either Payment by cash or Payment by credit/debit card, he/she can. The customer can also leave a feedback on his/her order and see the order progress/cart total meanwhile if he/she insists.

Manager Interface



The manager interface includes several administrator-level controls such as viewing Employee schedule and info, viewing orders, table status and edit table layout. In terms of flow, the manager first logs in and clocks in.

Server/Chef Interface



The chef/server interface is more less similar to the manager interface except for that the chef/server cannot view other employees' information and schedule. The chef/server cannot also edit table layout. The chef/server, like the manager has to log in and clock-in before accessing any tabs of the menu.

6 Design of Tests

6.1 Test Cases

Test Case Identifier: TC-1 **Use Case Tested:** UC-16 **Pass/Fail Criteria:** The test will pass if the user is able to successfully log in to the system. The test will fail if the user inputs an invalid username or password. **Input Data:** username, password	
Test Procedure:	**Expected Result:**
Step 1: User types in an invalid username and/or password. Step 2: User types in a valid username and password.	Server denies the log-in attempt. Message pops up, instructing the user to try again using a valid log-in. Server allows the log-in attempt. Depending on the user type, the system allows the user to access the specified interface.

Test Case Identifier: TC-2 **Use Case Tested:** UC-7 **Pass/Fail Criteria:** The test will pass if the user is able to successfully clock in and out for their shift and the server is able to verify their location. The test will fail if the user is unable to clock in or out, or the server is unable to verify the location. **Input Data:** Selecting either the clock-in or clock-out button	
Test Procedure:	**Expected Result:**
Step 1: User selects to clock-in or out outside of the restaurant. Step 2: User selects to clock-in or out within the restaurant.	Server allows the clock-in/out attempt. The employee's current status updates to keep track of whether they are currently clocked in/out. Database records that a clock-in/out attempt has been made in a foreign location, along with the current time, user, location, updated status, and a tag stating it was done outside of the restaurant. System allows the user to access the specified interface. Server allows the clock-in/out attempt. The employee's current status updates to keep track of whether they are currently clocked in/out. Database records the clock-in/out attempt, along with the current time, user, location, and updated

	status. Depending on the employee type, the system allows the user to access the specified interface.

Test Case Identifier: TC-3
Use Case Tested: UC-6
Pass/Fail Criteria: The test will pass if the user is able to successfully select different filters and the menu is able to adapt in real time and filter out the items that contain the selected ingredients. The test will fail if the menu does not properly remove the items containing the selected filters.
Input Data: selecting different filters checkboxes

Test Procedure:	**Expected Result:**
Step 1: User selects different filters in the menu options	Menu allows the filter options to be selected and the new get request to the server will not contain the items made with the filtered ingredients. The remaining items are re-organized, and the user is able to order any of the remaining items.
Step 2: User does not select any filters	Server get request contains all items available on the menu and the menu displays them.

Test Case Identifier: TC-4
Use Case Tested: UC-9
Pass/Fail Criteria: The test will pass if the user is able to successfully able to add items to the checkout cart and place the order
Input Data: item array

Test Procedure:	**Expected Result:**
Step 1: User selects different items and presses confirm order button	Server verifies that all items added to the cart are valid and the system prints “order successful”. The order is sent to the chef’s profile to be added to the order queue.
Step 2: User does not select any items and presses confirm order button	System recognizes the order as invalid and and indicates that the user must add items to the cart before placing an order.

Test Case Identifier: TC-5
Use Case Tested: UC -2
Pass/Fail Criteria: This test will pass if the user puts a valid credit or debit card into the program and it is recognized and accepted, or if an invalid credit card is inputted and is rejected. This test will fail if a valid card number is put in and rejected, or if an invalid card is accepted.
Input Data: Credit or debit card information

Test Procedure:	**Expected Result:**
Step 1: Type in incorrect card information. Step 2: Type in correct card information.	System indicates an invalid card number to the user, and then prompts the user to try again. The system processes the card number, accepts the card number and then deducts the charge from the users account and add the balance to the restaurant account.

Test Case Identifier: TC-6
Use Case Tested: UC-2 (Payment)
Pass/Fail Criteria: The test passes if the program properly sums the total amount of items, and applies tax and tip appropriately. The test case fails if the program cannot properly sum the items, apply tax or tip.
Input Data: User selects which items of food that they wish to add to the cart and input what tip they would like to add

Test Procedure:	**Expected Result:**
Step 1: User finishes selecting the desired items. Step 2: Automatically calculate appropriate total. Step 3: Manually calculate the total and compare the results with the program.	System will keep track of the selected items and their prices. System will add the price of the items together, calculate and add tax, and add tip in order to formulate the total. Person must calculate the total in order to check whether or not the system is doing it correctly.

Test Case Identifier: TC-7
Use Case Tested: UC-11
Pass/Fail Criteria: The tests passes if the program successfully calculates the proper amount of reward points for the user and adds it to their user account
Input Data: Customer Attendance

Test Procedure:	Expected Result:
Step 1: User must have an account or create one.	User has an fully functioning account.
Step 2: User must log into their account in order to have points awarded to them.	User successfully logs in.
Step 3: Manually calculate the total amount of points that they should have and compare the results with the program.	The program is successfully able to add the proper amount of new reward points to the existing balance.

Test Case Identifier: TC-8
Use Case Tested: UC-11
Pass/Fail Criteria: The tests passes if the program successfully calculates the proper amount of reward points for the user and adds it to their user account
Input Data: Customer Attendance

Test Procedure:	Expected Result:
Step 1: User must have an account or create one.	User has an fully functioning account.
Step 2: User must log into their account in order to have points awarded to them.	User successfully logs in.
Step 3: System calculate the appropriate amount of total points the user has.	The program is successfully able to add the proper amount of new reward points to the existing balance.
Step 4: Manually calculate the total amount of points that they should have and compare the results with the program.	Compare correct answer to the answer that the program calculated.

Test Case Identifier: TC-9
Use Case Tested: UC-12
Pass/Fail Criteria: The system passes if the user is able to successfully redeem their reward points on a reward that they have enough points for. The system fails if the user is able to get a reward that they did not have

enough points for, or if they are denied a reward that they should have been rewarded because they had sufficient points. **Input Data:** User input, the user must select that they wish to redeem their reward at that point in time.	
Test Procedure:	**Expected Result:**

Test Case Identifier: TC-10 **Use Case Tested:** UC-12 **Pass/Fail Criteria:** The system passes if the user is able to successfully apply their reward to their order, ie:if the reward was a free beverage the user should not be charged for that beverage. The test fails if the award is not properly applied to the bill	
Test Procedure:	**Expected Result:**
Step 1: The user must select the reward that they want to redeem and must meet the criteria for that reward. Step 2: The user goes to pay his bill. Step 3: The user pays his bill.	The user successfully selects that reward that they would like to get with their redeemed points. The system displays the corresponding screens and allows the user to input his payment information. The user is successfully able to pay the bill and did not have to pay for the item that was awarded.

Test Case Identifier: TC-11 **Use Case Tested:** UC-5 **Pass/Fail Criteria:** The test will pass if the user is able to successfully rate the food by selecting how many stars it deserves. The test fails if this is not so.	
Test Procedure:	**Expected Result:**
Step 1: User is on payment screen and user will be able to select how many stars the meal was.	When the amount of stars are selected then the stars should turn a solid color to indicate that was a star selected.

6.2 Test Coverage

All of our test cases cover all of the essential classes that are necessary to the operation of TurboYums. As we develop more and more of our classes and methods, we will add and adjust test cases as needed. Testing will be done for as many possible cases that a class could go through. The test procedures for the classes will have the format of both a pass procedure and a fail procedure in response to a user input. If the test fails due to a faulty user input, the system will prompt the user to try again and correct the input. An example of a test is testing the login screen, where if the user inputs username and/or password incorrectly, the system will have the user enter their information again. This fail case will repeat until the user inputs their correct information, at which point the user can login and start their shift. There are test cases for inputs from customers, employees, managers. Some test cases for the customers could be when they are creating an order. There could be a test case that makes sure if a customer wants to remove an item from an empty order it should display an error on the screen. Test cases for an employee trying to modify the table status could be that an error shows up if an employee is trying to clean a table that current has customers sitting at it. Our test cases will be specific so that if can cover all possible cases. We want our app to be efficient as possible with no errors which is why we will have test cases to show that it works properly.

6.3 Integration Testing

We decided on utilizing the strategy of bottom up testing. Bottom-up testing is an approach to integrated testing in that the lowest level components are tested first, and those components are used to help test the higher level components, this ensures that the building blocks of the code operate as needed before using them in other sections of the code. With this approach, we will be able to test what we are working on as we complete the different modules, since the lower level modules and components will be completed first and are part of the foundation of the higher-level components. This method of testing is the most appropriate for our project because if we were to choose a different method of testing, it would be much more difficult to understand what the cause of the bug is, whether the issue is coming from the interaction and integration of the different code components or if the problem is with how the classes are fundamentally designed and coded.

With bottom up testing we will be able to isolate where the problem is coming from because we will ensure the parts work properly prior to integration. By understanding the relationships between the objects in the system, the bottom up testing approach is more efficient and straightforward in that you can quickly narrow down where the problem lies prepping it for remedy.

A concrete way of representing the components of our system and how they would relate in this context would be that we test each of the employees independent and personalized job tasks in the application. For example, for the chef, we would test the implementation of the queuing system and make sure that it updates the server when the chef proclaims that a dish is cooked and ready to serve. After going through the individual functions, we test the features which call for interactions between more than one object or class.

7 Project Management and Plan of Work

7.1 Merging the Contributions from Individual Team Members

In order to maintain consistency throughout the report, each individual member was granted access to a Google Drive folder, containing each report and resources. This way, each member can see the entirety of the report while it is still being actively updated and worked on. To merge individual contributions of team members when it came to coding, our group utilized a shared GitHub repository, each feature oriented mini group branched the repository, and worked on implementing their features in that branch. Each group stayed in sync by continuously committing and pulling to the same branch. Upon completing pieces of functionality, a pull request will be made to merge those changes into the master branch and reviewed by the entire group. When completing the written parts of the reports, we were able to maintain overall consistency also by having multiple weekly meetings in person and online.

One problem we encountered when writing our reports is keeping our overall report format consistent. We had noticed that some headings were different and fonts/font sizes were varying. To fix this, we came up with a general format to stick to and we would all proof-read and compare our format to the other members' formats. Also, whichever member would be turning in the report each week would look it over and adjust any inconsistencies. Another issue was that our group has so many features we would like to implement that it can be easy to get overwhelmed. To solve this issue, we split our large group into three subgroups, which would specifically deal with tasks pertaining to their overall goal. We had a group working on the menu, a group working on the employee portal, and a group working on payment and the rewards system. This was done to divide the tasks evenly and by similarity, making it easier to get things done.

7.2 Project Coordination and Progress Report

We are currently focusing on UC-2 (Payment), UC-3 (View Menu), UC-5 (Rate Food), UC-6 (Food Filters), UC-7 (Clocking In/Out), UC-9 (Placing an Order), UC-11 (Earning Rewards), and UC-12 (Redeeming Rewards), which are all in early stages of implementation. Each of these use cases has been assigned to one of our smaller teams who specialize in each.

A database has been created for the application using MySQL and some of the individual tables for the classes have been created and added to the database. For the use cases mentioned, the classes that they require to operate have begun to be implemented as well as some of the more basic methods for a handful of the classes. For UC-9 both the user class, which is a class that contains all of the attributes and methods that any of the users of the software could potentially need and use, and the paymentMethod class, which is a class that will keep track of the type of payment made (cash or card) and if done by card will contain and store the card information, have been created, but have not yet been given all of the methods that they require to be fully functioning. The needed

tables for these two classes are created within the database. There is also in works of a table made for ingredients in the database to use for menu and placing order.

For UC-7 (Clocking In/Out), we have developed a class that keeps track of all information that is vital to the action of an employee clocking in/out. There currently exists variables within the class to keep track of the employee's unique identifier (employee id), the time of when they have clocked, their current clocking status, and whether or not their last clock attempt was on site.

For UC-4 (View Menu), an item class which represents a food selection available at the given restaurant using our software. It contains the item name, price, ingredients, and description has been created as the building block of the menu which will be necessary for all future classes pertaining to the menu.

For UC-9 (Placing an Order) an order class has been developed to represent the customer's order. This class utilizes the item class mentioned above when calculating the total price of an order and holding a list of the items selected by the customer. It also has an attribute called special request so that any need that the customer wishes to express that is not available through our interface can be entered on the tablet.

We have two designated meeting times a week (Wednesdays 4:40 PM and Thursdays 3:00 PM) as well as spontaneous meetings throughout the week when everyone is available in addition to our mandatory two meetings. We have approximately 2-3 voice chat meetings on evenings when people are available so we can keep in touch and update each other on our individual progress. In addition to this, we have established a group chat where we are kept in touch with each other 24/7 regarding important information and developments to the project. Lastly, working with more of our mini groups each group has their own group chat where they discuss about progress/code and have their own meeting times but usually break up into the mini groups at mandatory meetings.

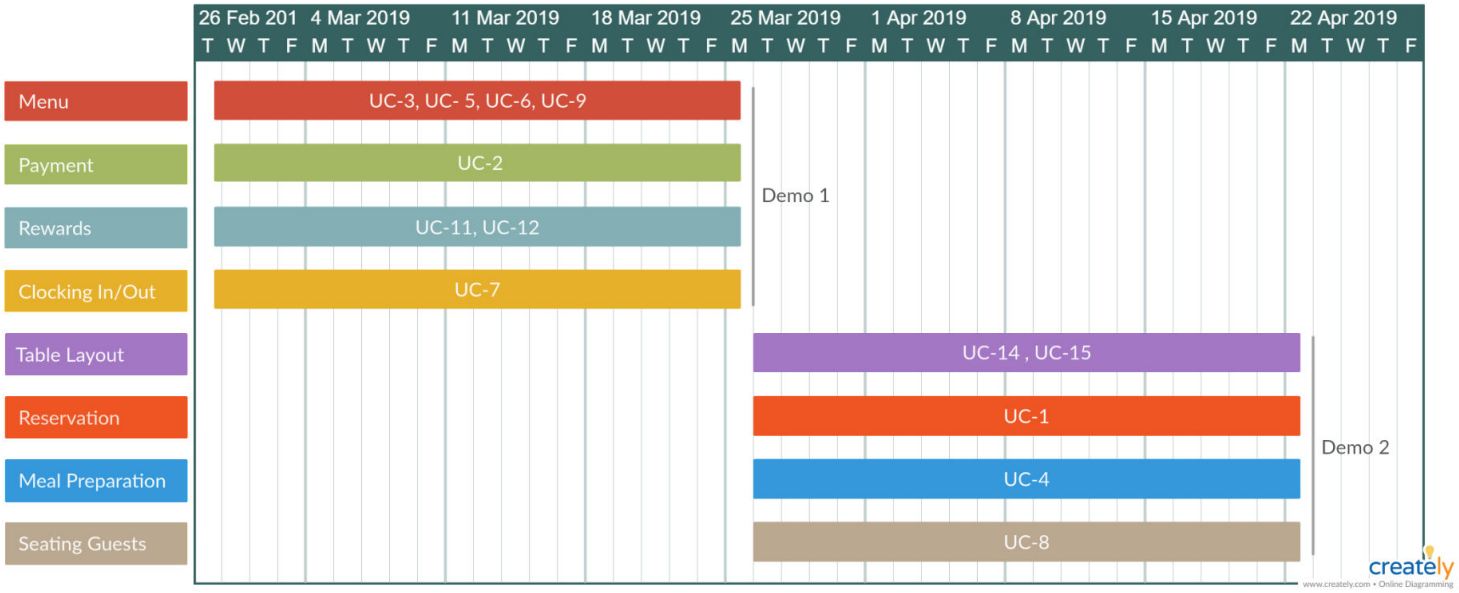
7.3 Plan of Work

As a group, we decided to split up the work of reports evenly as possible. Since report 1, we have assigned more specific tasks to our mini-groups, in order to focus on code specific features. After finishing report 2, we will continue testing and adding to our code, while continuing to look for ways to improve and will continue prepping for the demo.

Use Case	Functional Feature and Description	Start Date	End Date
UC-3 UC-5 UC-6 UC-9	Menu ● Food rating ● Food filter ● Placing an Order	02/26/19	03/25/19

UC-2	Payment	02/26/19	03/25/19
UC-11 UC-12	Earning/Redeeming Rewards	02/26/19	03/25/19
UC-7	Clocking In/Out ● User profile ● Employee portal	02/26/19	03/25/19
UC-14 UC-15	Table Layout ● Table adjustment ● Table selection ● Table Status	03/26/19	04/22/19
UC-1	Reservation	03/26/19	04/22/19
UC-4	Meal Prep	03/26/19	04/22/19
UC-8	Serving Guests	03/26/19	04/22/19

Plan of Work



7.4 Breakdown of Responsibilities

Outlined below are the teams, and the proposed work plan over the course of the next few weeks.

Team	Code Name	Members
Menu	Team A	Brandon, Holly & Roshni
Rewards and Payment	Team B	Dante, Suvranil & Ziad
Clock in/Clock out	Team C	Anthony, Hersh & Michelle

Short Term Plan of Work	Team
Create an interactive customer menu, in which a customer can order from	Team A
Create a menu filtering system	Team A
Create a rating food system	Team A
Implement a payment system	Team B
Create a rewards system	Team B
Create a standard Customer Profile to keep track of each customer's reward points	Team B
Implement a way for users to have the ability to log-in to our application	Team C
Create an Employee Portal for the employees	Team C
Implement an interface where employees can clock in and clock out	Team C

Team A:

Team A has been tasked with creating the menu system for the application which requires different classes such as a menu class, order class, and item class each containing their own attributes. These classes will allow the customers to browse the menu to see what foods and drinks the restaurant has. The customers can filter out certain foods that include ingredients specified. This will also allow customers to order food they would like to. Once customers are done eating, they will be allowed to rate the food they had. With all the coding that we do for these classes we plan to integrate our code with sequelize so that we can connect our application with our database.

Holly has created the item class which represents a food option and contains the item name, price, ingredients, description, rating, foodID, and item quantity. This class will be vital to the construction of the menu as it will be needed for almost all future menu classes and methods. Holly has also created the order class which represents the items that have been ordered from the menu. An order consists of the total price of the order, a list of ordered items, and special requests. She will later be working on removing food items that were added to the order that customers would like to remove from the order. The remove items will be a part of the menu class.

Roshni will be working on creating methods that will filter out items on the menu, this is part of the menu class. This method is the most important part of the menu as it is a unique option that users will be choosing which ingredients they are allergic/restrictions in food. Roshni will also be making the database for ingredients.

Brandon will be working on adding food items to cart. This will allow customers to look at the menu and then add food that they want to their order. This method is the add food method and will be part of our menu class. Without this method customers would not be able to add food to an order.

Team B:

Team B has been tasked with completing rewards and payment system for the application. As a result, this team will be implementing the following classes: Transaction, paymentMethod, and address since they are crucial to the two use cases.

Ziad and Dante work closely together and will do a combination of individual programming as well as peer coding in order to complete the three classes, transaction, paymentMethod, and address. The two will focus on creating the classes and their respective tables within the database and heavily focus on the backend components that are associated with the payment and rewards. They will also help with some of the of the interface and help Suvranil make decisions if help is needed.

Suvranil will play a smaller part in creating the classes, mostly to have an idea of the structure of the classes and database, but will have a large hand in creating the interface and be the connection between all of the different teams ensuring that the interfaces are consistent across the board.

Since the group only consists of three people, and Ziad and Dante will be closely working together while creating the classes and other modules, integration of the different component will not be very difficult and will likely be done as we are coding, with final integrations being completed by all three of us together.

Integration testing will be completed by Ziad and Dante, who will ensure that the different components are properly coming together.

Team C:

Team C has been tasked with completing a functioning Clock In/Out system for employees using the application. In order to implement this, the team must work on developing the Employee Portal object as well as the Employee Info object. The Employee Portal Object will contain information and methods that will be able to be called upon clocking in/out. The Employee Info object will contain information specific to an individual's clocking status.

Anthony has begun making a prototype of the class that will contain the data and methods to be used when an employee issues a clocking request. Those using the portal will be able to access the methods of the class to update their current clocking/working status.

Michelle will work on coding the individual methods of our features, while making sure our code is successful, as well as consistent with the other groups that are coding the other features. Michelle will also discuss with Suvranil from Team B to make decisions on the specific Employee Interfaces and how they should appear.

Hersh will work on coding the individual methods of our features, while making sure our code is successful, as well as consistent with the other groups that are coding the other features.

8 References

Richards, Mark. “Software Architecture Patterns.” O'Reilly | Safari, O'Reilly Media, Inc., https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html

Marsic, Ivan. “Software Engineering.” *Professor Zoran Gajic - Home Page*, www.ece.rutgers.edu/~marsic/Teaching/SE/.

Hanov Solutions Inc., “WebSequenceDiagrams” https://www.websequencediagrams.com/

Dimitrovski, Stephan, et al. *Why W8*. 2018, *Why W8*. https://www.ece.rutgers.edu/~marsic/books/SE/projects/Restaurant/2018f-g4-report3.pdf

El Warraky, Omar, et al. “Food•E•Z.” *Google Sites*, sites.google.com/site/sefoodez/home. https://hub.packtpub.com/what-is-multi-layered-software-architecture/

“Distribution Dashboard | Android Developers.” Android Developers, developer.android.com/about/dashboards. https://developer.android.com/about/dashboards

“IOS.” Wikipedia, Wikimedia Foundation, 5 Mar. 2019, en.wikipedia.org/wiki/IOS. https://en.wikipedia.org/wiki/IOS