Software Engineering

14:332:452

Group #11

Project URL: https://sites.google.com/site/softwareengineeringspring2012/

Restaurant Automation

Report 3

May 4, 2012

Jazmin Garcia
Greg Paton
Eric Gilbert
Andrew Rapport
Vishal Shah
Damon Chow

Team Contributions

All team members contributed equally.

Summary of Changes

In terms of changes, the project objective changes throughout the course of the project. Our team had numerous ideas we wished to implement, but we ultimately decided to only implement a few so as to show some key features that were fully functioning instead of many that did not work that well.

Our use case descriptions changed drastically from the beginning in that in the First Report, we had a total of 12 formal use cases that we hoped to focus on. We quickly learned that that was much too high of a number. We limited the number of use cases to focus on for the continuation of our progress to almost half of our initially planned 12.

From the First Report, our group learned that we needed to focus on providing much more detail to the reports and that the project was not focused on simply coding. The key to success to the project was to focus on a small amount of use cases and provide a large amount of detail.

Initially, we had hoped to have a delivery boy actor. In the end, we chose to eliminate that actor and focus on the other main actors. Our group considered the delivery boy to be an addition to our project but not necessary. Much of what is included in the Future Work is part of our First Report. For example, implementing a table layout for the hostess and waiter interface so as to allow the hostess to have a dynamic view of the restaurant as well as move tables when necessary. Due to the issue of time, we eliminated some of our goals as well as use cases.

TABLE OF CONTENTS

I. CUSTOMER STATEMENT OF REQUIREMENTS

A. Problem Statement

Running a restaurant incurs a great deal of overhead. Anyone looking to open and run a successful restaurant in today's economy needs to minimize cost to stay competitive. Therefore, a modern system could be implemented that would automate tasks that were once heavily time consuming and thus would increase efficiency and reduce operating costs. An example of how the restaurant would become more efficient is through the minimization of customer order time, which would consequently increase the average frequency of customer turnovers. Customer turnover frequency is defined as the number of parties that come and leave the restaurant.

More specifically, there will exist Touch Screen Monitors with which customers will have the option of placing orders on. The monitors will assist in decreasing order placement time as mentioned previously. Another option will be Wireless Order Placement. Using a wireless iOS Device, a waiter will be able to place an order in the situation that a customer does not wish to make the order him or herself. Through these two options, we give the customer the ability of choice.

The business policies of the company are listed in the VI section.

Figure 1: iOS or Android Compatibility

System Network

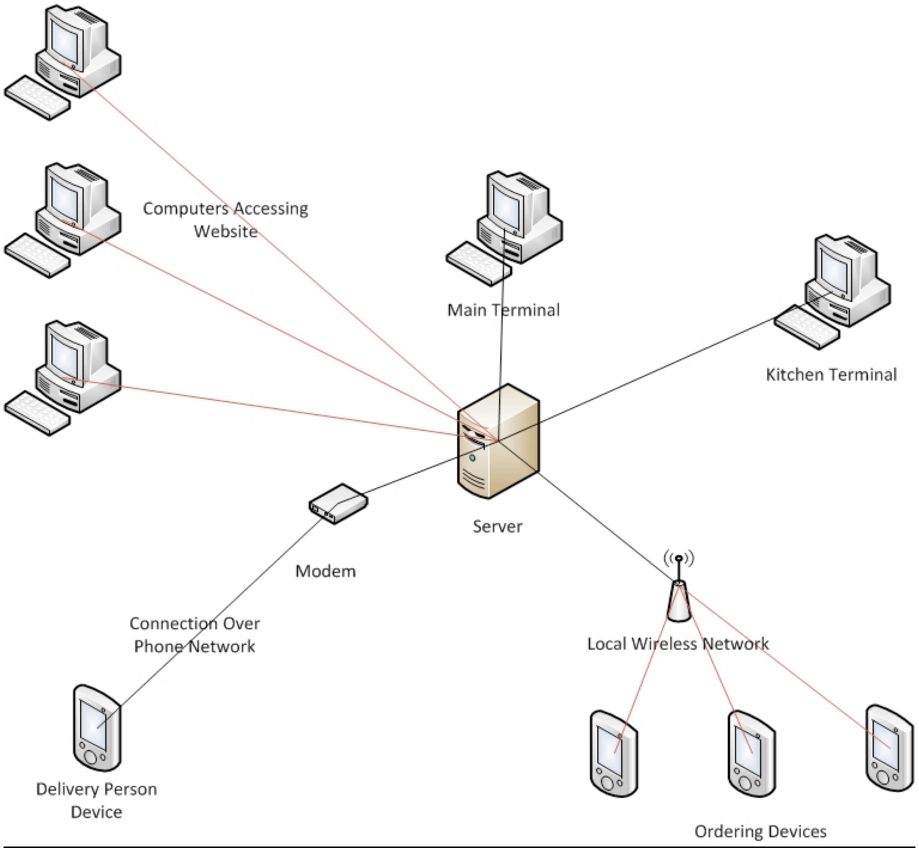


Figure 2 System Network

System Actors' Roles

Management

The system will give the manager access to statistical analysis tools such as the graphical and numerical breakdown of customer dining time, menu item popularity, the current stock of ingredients, and many more. Additionally, the system will give only management permission to do several tasks. Some of the tasks available to only the manager are listed below.

Employee management permissions:

- Add/Remove employees
- Customize employee permissions
- Create/Customize employee schedules
- Update restaurant floor plan layout
- Access/Modify database records

Host

The hostess is the first person a customer sees when they enter the restaurant. Being one of the most important assets to the restaurant in that the hostess is responsible for sitting the customer in a timely manner, there will be methods in the system that will assist the hostess. By having the ability to sit customers quickly and efficiently, the restaurant will be able to generate more revenue since there will be a high table turnover meaning that there is a high frequency of customers dining and leaving. Some of the abilities of the hostess are listed below.

- View the status of the tables in the restaurant
- Assign guests to a particular table given the number in the party
- Be able to give a wait time for the number of guests in the party
- Schedule reservations for customers

Waiter

The waiter will be the next person who the customer encounters with during the restaurant experience. The waiter will be assigned certain tables in the restaurant. This means that guests who sit in that particular section will be assigned to that waiter. The waiter will be responsible for taking orders for the guests in the situation that the guests do not wish to place the orders themselves via the computer system. Additionally, tables in the restaurant will have a status such as occupied, clean, or dirty. The hostess will be responsible for updating this status as well as being able to view the current status of the tables. Some of the abilities of the waiter are listed below.

- View/Update status of tables the server is assigned to
- Place/Edit/Remove order for specific tables

Bartender

- Place/Edit/Remove orders
- Receive drink orders placed by waiter/waitress

Chef

Once an order is placed either by the customer or by the waiter, the chef will be responsible for cooking the order. As soon as a chef has completed a particular order, the chef will press a button signifying that the order has been completed. A screen will be present, particularly for the chefs, which will display a maximum of 10 current uncompleted orders in the restaurant. When there are more than 10 open orders in the restaurant, the way the display will handle this is that once the first order is completed, the next order after the previous 10 will be moved to be displayed on the screen.

- View all present uncompleted order placed by waiter
- Update completion of order

Customer

The customer, as an actor, will have the option of placing an order. If the customers do not wish to place the order themselves, the customer can have the waiter place the order via the mobile application.

The Old System vs The New System

While the old pen and paper system may be suitable for smaller operations, any restaurant that receives a large throughput of customers can't be run using these outdated methods. The updated computer system increases efficiency, giving the restaurant a competitive edge by reducing overhead. By the use of this computer system, the table turnover rate will be drastically increased since placing an order via the system we will implement is much faster than the standard way of taking an order. When taking an order, a server is required to write down the order, walk to the computer terminal only available to employees (with the hope of not being interrupted on the way there), and then placing the order into the computer. The use of paper, in this system, is virtually eliminated. In conjunction with the ability of base ordering of ingredients on the computer via the analysis of stock and item popularity, operating costs are reduced. This system also allows the proprietor the option of running a "no waiter" restaurant that allows the customer to place their own order on devices embedded in the tables, reducing the number of employees and lowering cost. Efficiency is also seen with the hostess. Normally in a restaurant, a hostess to find where a party could be accommodated has to walk around the entire restaurant. With the new system, a hostess will be capable of viewing the status of all the tables in the restaurant; thus, reducing the amount of time guests must wait to be seated. Overall, implementing a computer system for a restaurant will result in higher efficiency and higher profits.

B. Glossary of Terms

- **Busboy** – Employee responsible for cleaning tables when a party leaves. Able to change table status from dirty to clean upon completion using the main terminal.
- **Chef** – Employee responsible for preparing food. Able to update the food order status using the kitchen terminal.
- Customer – A patron of the restaurant using the system.
- **Database** – Storage of pertinent restaurant data located on the server.
- **Host/Hostess** – Used interchangeably. Employee responsible for seating customers in a timely fashion. Able to change table status from clean to occupied using the main terminal.
- **Kitchen Terminal** – Touch screen computer located in the kitchen that updated the chef with orders. Also allows chef to update current orders.
- **Main Terminal** – Touch screen computer located in a central location in the restaurant. Available to all employees for clocking in/out.
- **Manager** – Employee responsible for running the restaurant. Given full permissions in the system. Has exclusive ability to view restaurant statistics, edit menu items, edit employee information, etc.
- **Manager's Terminal** – Computer located in the manager's office.
- **Menu** – A detailed list of all food items sold at the restaurant.
- **Mobile Device** – Touch screen, portable device used by waiters to place orders. May also be used my customer to place order and view order progress.
- **Party** – A group of customers seated together in the restaurant.
- **Payroll** – Hours worked by employees and hourly rate. Used to calculate salary.
- **Server** – Computer (located locally or at an external location) used to store the database and run various scripts.
- **Stock** – Amount of ingredients currently in the restaurants inventory.
- **Waiter** – Employee responsible for taking customer orders. Places orders using a mobile device, as well as handles payment from customer.

II. SYSTEM REQUIREMENTS

A. Enumerated Functional Requirements

	Rank	Description
REQ1	5	The system shall allow managers to control the payroll. This portion of the system should be accessible only by management via the web application. The payroll portion of the system should allow managers to add employees, remove employees, and update employee information in the database that is storing employee information. It should also allow managers to pay employees (discussed in more detail in REQ7). Please refer to the Business Policies in which employee removal is discussed further.
REQ2	3	The system should allow managers to view statistics showing the overall performance of the restaurant. This will include a display that allows managers to query about how much a particular employee has worked, figures showing the state of the inventory, what items are on the menu and how much of them have been sold, and overall financial figures on either a weekly or monthly basis. The statistical analysis will only be available to managers through the web application.
REQ3	2	The system shall have an inventory list that specifies what items are in stock, out of stock, and close to being out of stock. The inventory system shall be populated by the manager initially and whenever new items are delivered to the restaurant. It should then be automatically updated to reflect any changes to the menu that have been made. For example, if an order is placed, the inventory required to create that order should be taken into account and consequently update the inventory. Items will be removed from the menu when stock runs low. Additionally, the managers should have the capability to adjust inventory at the end of the day or week to reflect spoiled food or other unforeseen circumstances. Please reference the Mathematical Model section.
REQ4	4	The system shall allow the manager to alter menu items via the web application. When a menu item is added, the item should not be made available until all the components required to make the item is checked by management to be readily available. Once the inventory to create the new item is accounted for, the item will appear on the menu for the waiter mobile application as well as the customer desktop application.
REQ5	4	The system shall allow personnel to view the layout of the restaurant as a 2D diagram that mimics an aerial view of the floor. The layout will show tables and their configuration, and the tables will be colored to show the their status. Green will indicated an open table, yellow will indicate a vacant table that must be cleaned and preperad for another customer, and red will indicate and table currently in use. The host will be capable of marking a table red via her terminal application. The server will be capable of marking the table yellow via his mobile application. The busboy will be capable for marking a table green via his terminal application. The system will allow alteration of the floor plan by managers via a web application and a hostess via the hostess terminal application.

REQ6	2	The system shall allow managers, waiters, and hostesses to both physically move the tables in the restaurant and change the diagram showing table layout through their respective devices describe in the table layout application requirement.
REQ7	5	The system shall allow managers to record the time an employee has worked via the system. Employees shall be able to clock in and out of the system, with the time of work being logged. If an employee fails to clock out of the system after a maximum threshold is crossed, he will automatically be logged out. The maximum threshold will be the time that the particular employee was to work till. Any time past the threshold will require a manager to approve the overtime. This threshold, along with hourly pay and overtime pay, will be stored in a database along with the name, address, and social security number of the employee. The payroll should allow the manager to release payment calculated through hours logged and employee information once every two weeks via the web application. Upon payment, the employee hours shall be populated into a database for statistical analysis.
REQ8	5	The system shall allow both customers and waiters to place orders which will then be sent directly to the screens of the chef terminal in the kitchen. Customers will be capable of ordering through a computer terminal located at their table, while waiters will be able to place an order through the mobile device they carry on their person at all times. Alcoholic drink orders cannot be placed via the customer application. Servers must check for identification at the table prior to the successful order of alcohol.
REQ9	4	The system should populate delivery orders on the screens of the chef terminal and the deliveryman mobile application. The chefs can then remove the order from their screen once it has been prepared, and the deliveryman can remove the order from his mobile application when it has been delivered. If the order does not make it to the customer, he or she can call to complain or submit a complaint online via the web application.
REQ10	3	The system should allow customers to beckon a waiter via the computer terminal located at their table. To limit the complexity of the system and opportunity for customer tampering, the customer beckoning will simply show up on the table layout diagram as a dot layered on top of the red table. This can be viewed by anyone with access to the table layout diagram through the devices described in that requirement.
REQ11	5	The system shall have a menu screen on the computer terminal of customers for customer ordering. The menu screen should allow the customer to place their order through the customer computer terminal.
REQ12	1	The system should allow reservation to be made online via the web application. Online reservations will only be allowed if the reservation is placed 24 hours in advance of the time requested. Otherwise, the customer can call the host and the system should allow the host to input the reservation if it is available via the host terminal application. Regardless of online or phone reservation, the system shall prompt the customer or host respectively for name and phone number. If the customer is logged in via the web application, this information will be retained.
REQ13	1	The system should allow accounts to be made for future login by the same customer. This feature is predominantly for delivery order, so that customer information such as name, phone number, address, and credit card information can be retained. Thus, the customer won't be forced to re-enter information each time an order is placed.

REQ14	5	The system should allow customers to pay for their check at the table via their computer terminal. Payment information should be entered via a prompt on the screen. Only electronic payment will be allowed via the customer mobile application. The customer can alert the server either through the mobile application or personally if another payment method is desired. The server can make payment through the server mobile application (credit cards only) or terminal (all acceptable forms of payments).
REQ15	2	The system should allow the customer to create an order change within a particular time period. This time period is defined as the time up until the chefs begins making the order. The manager is the only person that could approve order cancelations. Please refer to the Business Policy section for more details on order cancelation.
REQ16	4	The system shall provide the host with the tools and information to seat patrons in a reasonable amount. A reasonable amount of time will be dependent on the current status of the restaurant. When it is busy, the wait time will be much longer than if it were slow. The system will allow the host to be able to see the current status of tables in the restaurant for a much faster and accurate seating time.
REQ17	5	The system shall have screens (controlled by a computer terminal) in the kitchen for chefs. On the screens will be the orders listed in the order that they came in. The screens will be run by the chef terminal application located on the computer. 10 orders will be able to fit on a screen.

B. Enumerated Non-Functional Requirements

Non-functional requirements are those requirements that articulate the goals of the software development beyond the core necessities of the project. Such pieces of information work to explain how choices have been made to best allow the program to make the jump from a testing environment into the wild. The five categories of non-functional requirements are functionality, usability, reliability, performance, and supportability.

1. Functionality
    a. Compartmentalization of the different roles of restaurant operations to provide targeted solutions to each employee
        i. Mobile application used for roles requiring movement, such as serving, delivering, and ordering food
        ii. Terminal application used for roles that don’t necessitate mobile technology solutions and offer backup to the mobile app
            1. Statistical package that interfaces with payroll, table management, and inventory to facilitate management decisions
            2. Interactive table layout manager for flexibility and better visualization of restaurant state
            3. Manager options.

b. System scalable and flexible enough to accommodate restaurants with different sizes, layouts, strategies, etc.

c. Strong and well-implemented software design to limit exposure to electronic attacks

d. Strong log-in procedures to prevent tampering and misuse of software

e. Limited access to parts of the system for each particular user

3. Usability

. Clean and intuitive user interfaces that requires only a small amount of hands-on training to use proficiently

a. Exciting graphical and technological experience to enhance dining and employee satisfaction

b. Limited reliance upon human inputs such as writing or changing orders

c. Minimal "clicks" to accomplish the necessary task

d. Well-documented analysis, design, and code to teach, improve, or expand the system

Reliability

. Goal is for the capability to operate the restaurant under all foreseeable conditions, including internet outage, power outage, or partial system failure (due to circumstances other than software issues)

a. Server located on-site at restaurant to ensure ongoing operations in the face of internet outages

b. On-site servers could connect to an even more centralized server for computing requirements and backup capabilities

c. Terminals provide an interface to all employees in the event the web or mobile applications are not available

d. **[TERMINALS CACHE DAILY ACTIVITY IN ADDITION TO PINGING THE SERVER TO PROVIDE LOCAL BACKUP IN THE CASE OF OUTAGES OR VIRUSES]**

e. Backup pen and paper always available for unlikely events such as total power outage to maintain order and proper book-keeping until normal operations resume

Performance

. Easily navigable interface, well-written code, and the current processing power of recommended hardware will ensure system speed

a. Interface will also limit number of clicks and integrate different parts of the system to make for efficient but powerful interaction with the software

b. Resource requirements slightly higher than that for standard restaurants. This includes computing power for the mobile applications to be used by servers and customers.

c. Excellent design and code implementation will ensure a responsive system for all parties.

d. System will utilize a wireless network connection to provide adequate throughput of data to the server. All employees working at one time will be capable of interacting with the system simultaneously(usually 5-10 servers at one time). This exceeds the requirements of even a large restaurant.

e. System availability will depend upon user in question.
    i. Mobile applications available to servers only when they have clocked in to a terminal for work
    ii. Terminals available during working hours of the restaurant
    iii. Server available all times except maintenance (> 99%)

f. System updates to occur when restaurant opens/closes, reservation made, employees logs in/out, table opens/closes, customer order placed/changed, and payroll release.

g. User interface provides confirmation screens and limited human input to increase accuracy of system. Solid design and implementation ensure data integrity in producing payroll reports, statistical analysis, or managing inventory.

Supportability

0. Testability: unit tests will be written as the software is developed to allow the testing process to accommodate changes in code.
1. Extensibility: the design and implementation of the code will be executed with readability and extensibility in mind.
2. Adaptability/compatibility: the technologies **[MYSQL, JAVA, iOS]** are nearly ubiquitous and can port to a multitude of systems.
3. Maintainability: the design and implementation of the code will allow for it to be easily read and updated. The software stack has also been chosen as one that will most likely be supported very well in the long term. Patches could be deployed via dynamically linked libraries.
4. Configurability: to be completed on an ad hoc basis by the developers
5. Portability: the system will not be designed with portability in mind. Because of the technologies chosen, however, it could potentially be ported to other operating systems. For now, it will be assumed that the restaurant computers will run **[WINDOWS]**, whether by the native operating system or by virtual machine.

	Rank	Description
Functionality		
REQ18	3	The system should be designed is such as way that it could be extended to a different restaurant with a different business model in less than a month
REQ19	3	The system should utilize technology that is flexible enough to work on various platforms and new hardware. In the event that the software does not work on a particular piece of hardware, it should take no more than a month to port the software to the different hardware configuration.
Usability		
REQ19	4	The various interfaces of the host terminal (table layout, reservation, waitlist) will be available in one click from the root host menu

REQ21	5	The UI of the chef terminal should always show the current order queue. The chef should be able to notify the server that food is ready in 1 click. The order queue should update automatically on the placement of an order or the completion of notification of food completion.
REQ22	5	The screens of the chef terminal should be in view of the chefs and text should be large enough to read from 10 ft. away.
REQ23	4	There should be enough server terminals such that it doesn't take a server longer than 15 seconds to reach his terminal from the farthest table he is serving.
REQ24	2	The manager should be capable of adding, removing, or updating employee information in less than 3 minutes.
REQ25	2	The manager should be capable of viewing pay information and releasing payment to employees in less than 3 minutes.
REQ26	5	The user interfaces shall be aesthetically pleasing both to customers and to employees. Aesthetically pleasing will be generally defined as having a favorable appearance to 80% of 10 customers and 10 employees asked about the appearance of the program at random.
Reliability		
REQ27	4	The hostess terminal shall be reliable to the extent that it does not crash more than once every 2 months.
REQ28	5	The chef terminal shall be reliable to the extent that it does not crash more than once every 2 months
REQ29	5	The customer mobile application should reliable to the extent that it does not crash more than once every 6 months. Because the customer application will connect to a local wireless router, lack of internet access should not serve as an impediment to the customer mobile application.
REQ30	3	The web application should be reliable to the extent that it does not crash more than once per month.
REQ31	5	The system shall function in the absence of an internet connection. While the web application would not function in such a case, all the standard operations of the restaurant could occur. Once internet access is restored, the manager could update any necessary parts of the system.
REQ32	2	The server shall have the built-in capability such that networking it with a central server or backing up any data on it would take less than 3 days.
REQ33	1	The server shall have the capability of updating code dynamically to keep uptime above 99% in the event that updates or fixes are required.
Performance		
REQ34	4	There should be a delay of no longer than 400ms for all operations on all devices except for statistical queries run from the web application on the database. A 1 second delay is acceptable for database queries with mathematical calculations.
Supportability		
REQ35	3	The system design and implementation should be documented and simple to the extent that a capable IT company could perform updates and fixes within 3 days

C. On-Screen Appearance Requirements

Since our project is a multi-platform design (android-based, computer terminal-based, and web-based), we will have many on screen requirements we have to handle. Designing in just android brings up many requirements. Being that android is a multi-device platform, we will have to design our UI in a way that will accommodate all different types of devices. For example, there are android devices in the form of tablets and phones that can be used with our system. Both android tablets and phones comes in a wide variety of screen sizes and pixel densities. Since it easy to port an Android app to run on a PC(and thus a web interface) using porting programs such as "YouWave" , we will only focus on the on screen requirements for Android devices. As well as hardware requirements, we also have to address usability requirements. Our system should not only have a clean and polished look, but also be extremely easy to use for experienced users and new users alike.

Android suggests making three different versions of your UI to accommodate screens that are “Low Density”, “Medium Density”, and “Large Density”. But what about the different sized screens within these categories? The best way to deal with this problem is so design “Fluid” layouts, meaning that no matter what size screen one has, the program will automatically re-size the UI to fit the screen. Luckily, the Android framework has this feature built-in. There are various kind of layouts available (LinearLayout, FrameLayout etc.) that will gracefully fit images based on screen resolution of the device.

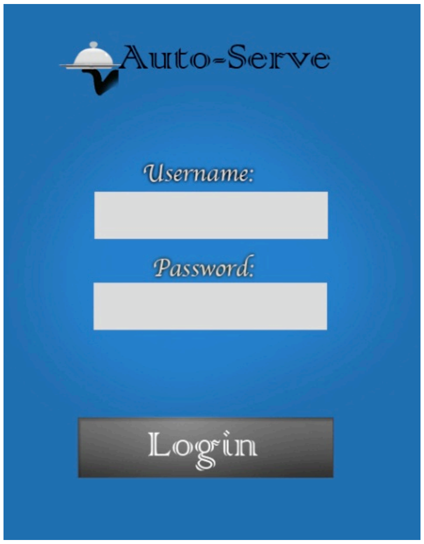
To accommodate on screen usability requirements, we will design a UI that will require a minimal amount of button presses to get to important features (5 or less), an intuitive design flow, big buttons that can be ready from a distance of at least 2 feet away, and buttons that have icons that show what the feature does in the case that someone that is not using the system does not speak English as their first language.

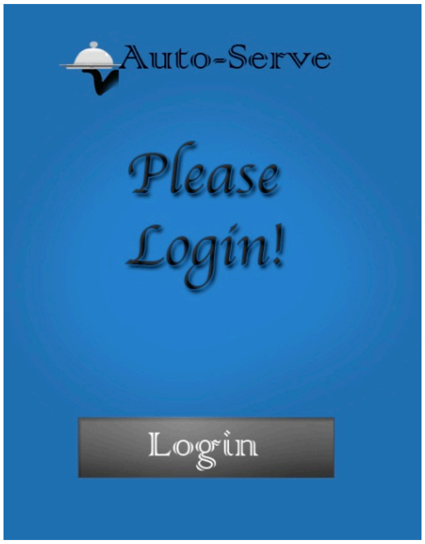
Below is a table with a prioritized table of on screen requirements:

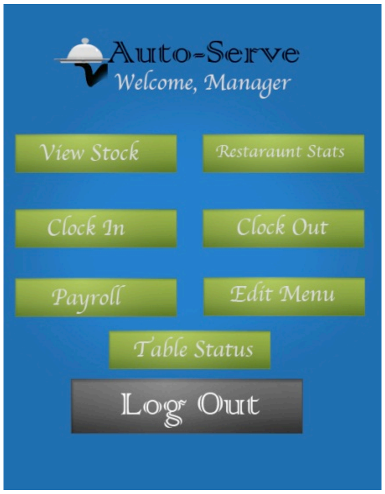
Identifier	**Priority**	**Requirement**
UIREQ-36	5	The UI shall be able to re-size itself to be able to fit on different sized screens and devices.
UIREQ-37	5	The UI shall have multiple sized versions for different screen densities.
UIREQ-38	5	The UI shall be designed in a way that is easily ported to a PC and web-based application.
UIREQ-39	4	The UI shall take no more than 5 button presses (besides typing characters to log in) to reach desired program features.
UIREQ-40	3	The UI should have buttons that can be seen from at least 2 feet away
UIREQ-41	4	The UI shall have buttons with icons that easily describe program

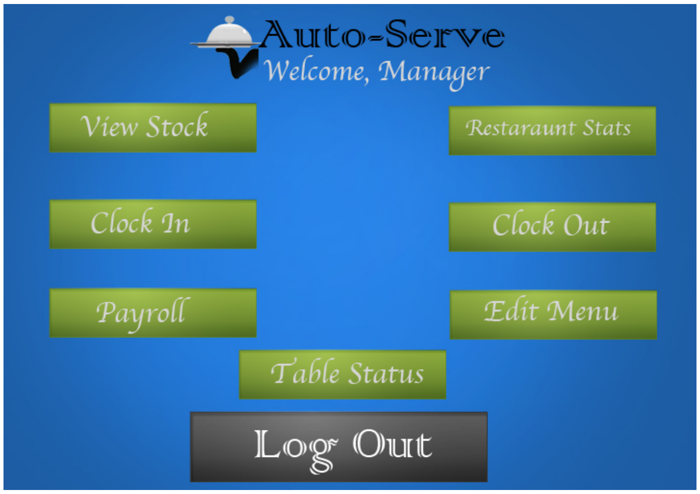
		features to users that do not speak English as a primary language.
UIREQ-42	3	The UI shall have an intuitive flow that can be easily used by new users as well as experienced users.
UIREQ-43	1	The UI should have a version in English as well as Spanish.

Below are some screen shots showing designs on how we plan on overcoming UIREQ-2, UIREQ-4, and UIREQ-5. These images depict a UI that has big, easily read buttons, a navigation path that only take 3 button presses to choose a feature from the options menu, and multiple sized screens to accommodate a phone and a table android device.









III. FUNCTIONAL REQUIREMENTS SPECIFICATION

A. Stakeholders

Usually, programming projects have five main stakeholders: the project managers, software testers, program architects and developers, system analysts, and lastly clients. Clients are defined as both employees as well as the company that will purchase the system. For the purpose of this project, our group as a whole will assume the role of project managers, software testers, program architects and developers and system analysts. Thus, the main focus for our stakeholders will be the clients.

Our stakeholders are divided into two main categories:

1. **Actors**
   This includes all of our program actors: customer, waiter, hostess, manager, and delivery boy. These actors will be interacting directly with our system.

2. **The Company that will purchase the system**
   This would include the head of the restaurant/restaurant chain. These people generally will not be interacting directly with our system.

B. Actors and Goals

Initiating Actors:

1. Customer
   a. This actor places a new order through the system at the table. The order that is placed is relayed to the kitchen (UC-4: PlaceOrder)
   b. This actor can edit a current order as long as the order has not been completed (UC-5: EditOrder)
   c. This actor can request a reservation (UC-1: MakeReservation)
   d. This actor, after have eaten meal, can pay their bill (UC-7: PayBill)
2. Chef
   a. This actor clocks in and clocks out when entering/leaving work (UC-25: ClockIn, UC-26: ClockOut)
   b. This actor has access to the statistics of the inventory of the restaurant (UC-16: ViewStockStats)
   c. This actor completes the order placed by the customer (UC-8: CookOrder)
3. Waiter
   a. This actor clocks in and clocks out when entering/leaving work (UC-25: ClockIn, UC-26: ClockOut)
   b. This actor logs into the system to their specific interface (UC-31: AuthenticateUser)
   c. This actor adds/removes items from an order (UC-5: EditOrder)
   d. This actor places orders for customers when a customer is having difficulty (UC-4: PlaceOrder)

   e. This actor collects payment from guests at a table when the guests are ready to pay their bill (UC-7: PayBill)
   f. This actor can change the status of a table when a table is either occupied or empty (UC-11: TableOccupied, UC-12: TableClean, UC-13: TableDirty)
4. Host
   a. This actor clocks in and clocks out when entering/leaving work (UC-25: ClockIn, UC-26: ClockOut)
   b. This actor logs into the system to their specific interface (UC-31: AuthenticateUser)
   c. This actor seats a Customer at a table when requested by the customer (UC-28: SeatCustomer)
   d. This actor can alter the floor plan of the restaurant for situations such as large parties (EditLayout)
   e. This actor can view the current status of tables in the restaurant and see whether they are occupied, dirty, or clean (UC-2: CheckTableStatus)

5. Manager
   a. This actor clocks in and clocks out when entering/leaving work (UC-25: ClockIn, UC-26: ClockOut)
   b. This actor logs into the system to their specific interface (UC-31: AuthenticateUser)
   c. This actor adds/removes employees from the database as well as updating an employee’s information (UC-22: AddEmployee, UC-23: RemoveEmployee, UC-24: UpdateEmployee)
   d. This actor updates the menu by adding/removing items (UC-18: AddtoMenu, UC-19: DeleteFromMenu)
   e. This actor has access to the statistics of the inventory of the restaurant (UC-16: ViewStockStatus)
   f. This actor can view the restaurant statistics, which includes financial statistics (UC-17: ViewStats)
   g. This actor can alter the floor plan of the restaurant for situations such as large parties (UC-3: EditLayout)
   h. This actor can issue payment to employees (UC-20, ReleasePayment)
6. Delivery Boy
   a. This actor clocks in and clocks out when entering/leaving work (UC-25: ClockIn, UC-26: ClockOut)
   b. This actor logs into the system to their specific interface (UC-31: AuthenticateUser)
   c. This actor notifies the system when a delivery has been completed (UC-21: CompletedDelivery)

Participating Actors:

1. Chef:
    a. Once the order of a customer has been placed, the order shall be placed on a queue consisting of all current open orders. This actor will prepare the food that is currently on the queue with those that have been placed first being prepared first.
2. Server
    a. This actor brings food and drinks to a table when these products are ready for the customer.
    b. This actor also responds to Customer's calls which occurs when a Customer would prefer to place the order with the server.
3. Busboy
    a. This actor is responsible for cleaning tables at the restaurant once the server marks the table as dirty.

C. Use Cases

I. *CASUAL DESCRIPTION*

USE CASE	**DESCRIPTION**
MakeReservation (UC-1)	Allows a *customer/hostess* to place a reservation.
CheckTableStatus (UC-2)	Allows *all personnel* to see whether a restaurant table is occupied, clean or dirty.
EditLayout (UC-3)	Allows the *hostess/manager* to change the layout of restaurant tables, including the combination of several tables into one to accommodate larger groups.
PlaceOrder (UC-4)	The *waiter* may place an order through the mobile application system for their respective tables.
EditOrder (UC-5)	Allows a *customer/waiter* to add/modify an order that has been placed depending on whether or not the order has been placed.
CancelOrder (UC-6)	A *waiter* cancels an order that has already been placed depending on whether or not the order has been placed.

PayBill (UC-7)	Allows a *customer* to beckon the *waiter* for a check/receipt.
CookOrder (UC-8)	The *chef* prepares the food according the customers' orders.
(UC-9)	Intentionally omitted to preserve numbering system
(UC-10)	Intentionally omitted to preserve numbering system
TableOccupied (UC-11)	Allows the *hostess* to denote a table occupied.
TableClean (UC-12)	Allows the *hostess/busboy* to change the table status to clean to denote a clean table ready for new customers.
TableDirty (UC-13)	Allows the *hostess/waiter* to change the table status to dirty and notify the *busboy*.
NewItem (UC-14)	Allows the *manager* to create a new inventory item.
DeleteItem (UC-15)	Allows the *manager* to delete an item from the inventory.
ViewStockStats (UC-16)	Allows the *chef/manager* to check the current inventory in the restaurant.
ViewStats (UC-17)	Allows the *manager* to see the financial and ordering statistics.
AddtoMenu (UC-18)	Allows the *manager* to add an item to the restaurant menu.
DeleteFromMenu (UC-19)	Allows the *manager* to delete an item from the restaurant menu.
ReleasePayment (UC-20)	Outputs the pay of *restaurant personnel* on a monthly basis, resetting hours worked back to 0 after payout.
CompletedDelivery (UC-21)	Once the *delivery boy* finishes a delivery, the order is removed from the list of queued items.

AddEmployee (UC-22)	Allows the *manager* to create a new employee account and place it in the database.
RemoveEmployee (UC-23)	Allows the *manager* to delete an employee account from the database.
UpdateEmployee (UC-24)	All the *manager* to update employee information in the database
ClockIn (UC-25)	The *restaurant personnel* notify the system of when they started working.
ClockOut (UC-26)	The *restaurant personnel* notify the system of their departure time and adds the total hours clocked that day to their monthly totals.
RequestTable (UC-27)	The *customer* asks to be seated at a certain table or for an open table.
SeatCustomer (UC-28)	The *hostess* assigns a table to the *customer*.
GetPartySize (UC-29)	Notifies the *hostess* of the number of people to arrive or be seated with a *customer's* request.
CleanTable (UC-30)	The *busboy* is responsible for cleaning a particular table when a party gets up and leaves the restaurant.
AuthenticateUser (UC-31)	The *hostess, waiter,* and *manager* will be verified that they have the authorization to enter a particular interface.

II. USE CASE DIAGRAM

The use case diagram displays the relationships between the various actors in the system and the use cases. The use cases are connected to any characters that they require as well as other use cases that are necessary for their completion. The manager is able to perform all of the use cases. However, to simplify the appearance of the diagram, an arrow structure was used to show that the manager is capable of performing any of the abilities the other employees can perform. This kept a cleaner appearance while still conveying the dynamic of the system.

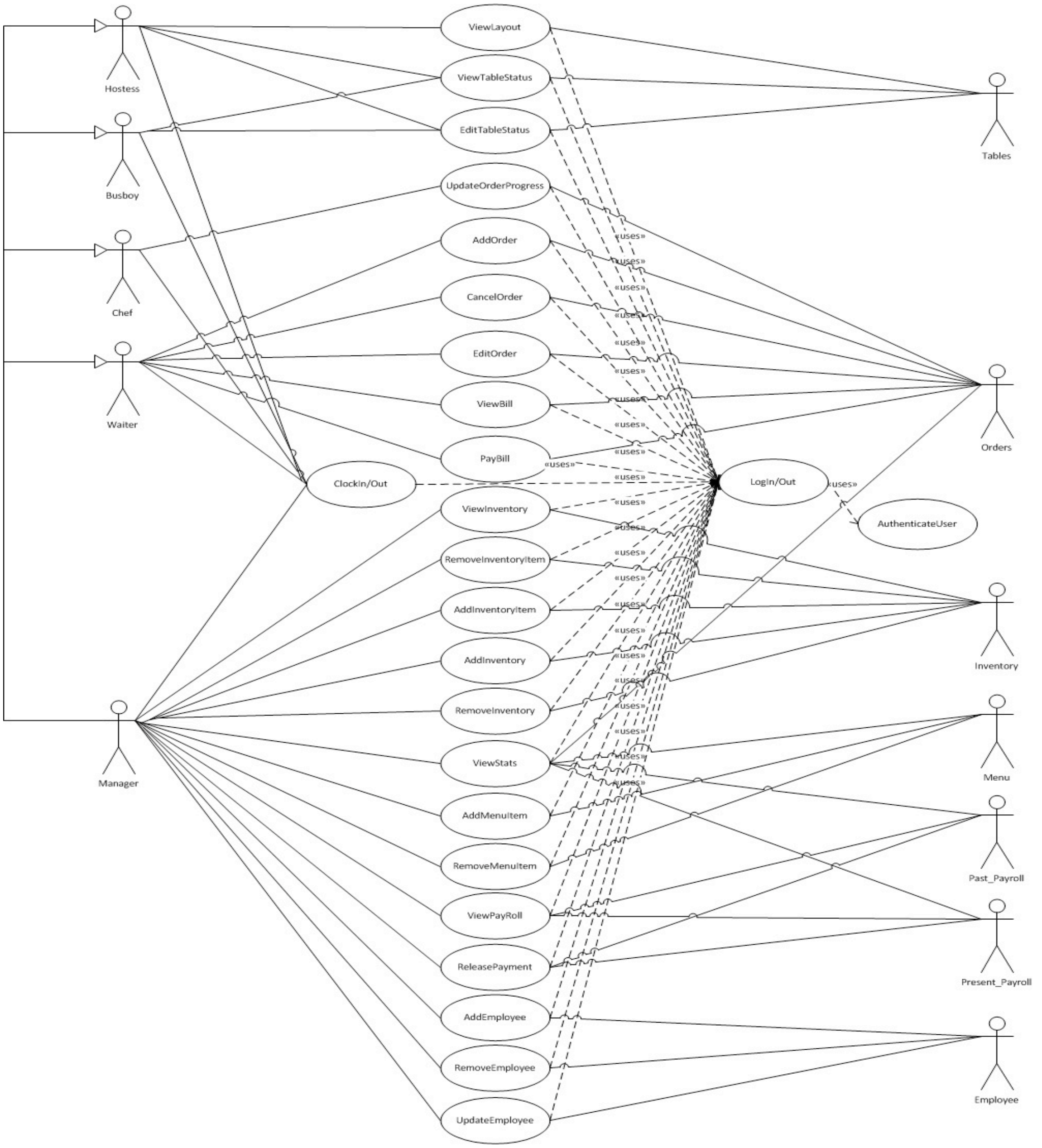


Figure 3 Use Case Diagram

III. FULLY DRESSED DESCRIPTION

Use Case UC-17	**ViewStockStats**
Related Requirements:	REQ1, REQ2, REQ3, REQ4, REQ7
Initiating Actor:	Any of: Manager, Chef
Actor's Goal:	To be able to view the statistics of individual menu items. Specifically, how much of each item is used when it is ordered as well as the amount of each item currently available to the restaurant.
Participating Actors:	Any of: Manager, Chef
Precondition:	There must be items on the menu for which to view the current stock of the ingredients that make up the menu items.
Postcondition:	The statistics of the inventory are successfully viewed, and all the inventory are stocked.
Flow of Events for Main Success Scenario:	→ 1. Manager logs in to system 2. include::*AuthenticateUser* → 3. Manager chooses from menu "View Restaraunt Statistics" ← 4. System displays a list of the current available inventory of the restaurant. → 5. Manager chooses an inventory item from the list to view more detailed information.
Flow of Events for Extensions (Alternate Scenarios):	2a. *AuthenticateUser* unsuccessful. → 1. System signals *AuthenticateUser* unsuccessful. ← 2. Manager re-enters password. 5a. A particular inventory item from the list states that the item is out of stock. ← 1. Manager orders more of the stock item through the system.

Use Case UC-4	**PlaceOrder**
Related Requirements:	REQ8, REQ17
Initiating Actor:	Any of: Customer, Waiter
Actor’s Goal:	To record the orders of guests to be relayed to the kitchen for preparation.
Participating Actors:	Any of: Chef
Precondition:	The table has been designated as occupied as well as having the specified number of guests at the table.
Postcondition:	An order is placed, and the chef is notified.
Flow of Events for Main Success Scenario:	→ 1. Customer/Waiter selects “Add Item” on system interface. 2. include::*AuthenticateUser* ← 3. System provides the actor with an electronic menu. → 4. Customer/Waiter selects an item on the menu. ← 5. System provides the option of editing the details of the particular item. → 6. Customer/Waiter placed the order, and the item is added to the bill under the particular guest that ordered the item. ← 7. System notifies the Chef that the order is placed.
Flow of Events for Extensions (Alternate Scenarios):	3a. Customer asks Waiter for help on ordering → 1. Waiter orders for customer on a portable interface. ← 2. System relays the ordered item to the kitchen/Chef. 3. Same as step 6 above. 6a. Item is out of stock. ← 1. System notifies actor that item is unavailable and shows the main menu so that the customer can choose a

	different menu item. → 2. Customer/Waiter selects "Add Item" on system interface. 3. Same as step 4. 6b. Customer/Waiter cancels order prior to the chef beginning to prepare the item. ← 1. System updates the bill of the guests so that the order canceled is now removed. ← 2. System returns to the main menu. → 3. Customer/Waiter selects "Add Item" on the system interface to choose a different menu item. 4. Same as step 4.

Use Case UC-2	CheckTablesStatus
Related Requirements:	REQ5, REQ6
Initiating Actor:	Any of: Hostess
Actor's Goal:	Determine what tables are clean, occupied, dirty, or reserved.
Participating Actors:	Any of: Waiter
Precondition:	There are available tables in the restaurant.
Postcondition:	System returned status on a particular table.
Flow of Events for Main Success Scenario:	→ 1. Customer requests a table by walking into the restaurant. 2. include::*RequestTable* → 3. Hostess checks the status of the tables in the restaurant. 4. include::*AuthenticateUser* ← 5. System returns the status of the tables in the restaurant. 6. include::*SitCustomer*
Flow of Events for Extensions (Alternate Scenarios):	5a. System returns that there are no tables currently available. → 1. Hostess waits a period of 5 to 10 minutes to check the table status of the tables in the restaurant. 2. Same as 3.

Use Case UC-7	**PayBill**
Related Requirements:	REQ14
Initiating Actor:	Any of: Customer, Waiter
Actor's Goal:	Allows the customer to pay for their total bill.
Participating Actors:	Any of: Waiter
Precondition:	The bill correctly corresponds to the correct table.
Postcondition:	The bill has been successfully and precisely paid.
Flow of Events for Main Success Scenario:	→ 1. Customer has reviewed the bill's accuracy and now wishes to pay the bill. → 2. If paying by cash, the waiter makes sure it is the correct amount. If paying by debit or credit, the waiter makes sure the transaction is successful. 3. include::*AuthenticateUser* ← 4. System asks Waiter to input the table the bill corresponds to as well as the method of payment.
Flow of Events for Extensions (Alternate Scenarios):	2a. The transaction done by debit/credit was unsuccessful. ← 1. System asks the Waiter to try again or use a different method of payment. 2. Same as 4.

IV. TRACEABILITY MATRIX

Req	PW	UC-1	UC-2	UC-3	UC-4	UC-5	UC-6	UC-7	UC-8	UC-9	UC-10	UC-11	UC-12	UC-13	UC-14	UC-15	UC-16	UC-17	UC-18	UC-19	UC-20	UC-21	UC-22	UC-23	UC-24	UC-25
1	5																X	X	X	X	X				X	X
2	3													X											X	X
3	2								X	X	X	X	X												X	X
4	4														X	X									X	X
5	5																X	X				X			X	X
6	5	X						X																	X	X
7	2		X	X		X	X																		X	X
8	4				X																		X	X	X	X
9	5							X																	X	X

D. SYSTEM SEQUENCE DIAGRAMS

Authenticate User

The following diagram displays the sequence for the main success scenario of the use case AuthenticateUser. This includes the usage from all restaurant personnel to initially login with a designated username and password to be checked with the stored information in the database via an "AuthenticationRequest." The alternate scenario for this use case would have all the same sequence of steps with the exception of the final return message from the system being an indication of a failed login attempt. As a result, the alternate scenario is a rather trivial case with the main success scenario given below and will not be diagramed below.

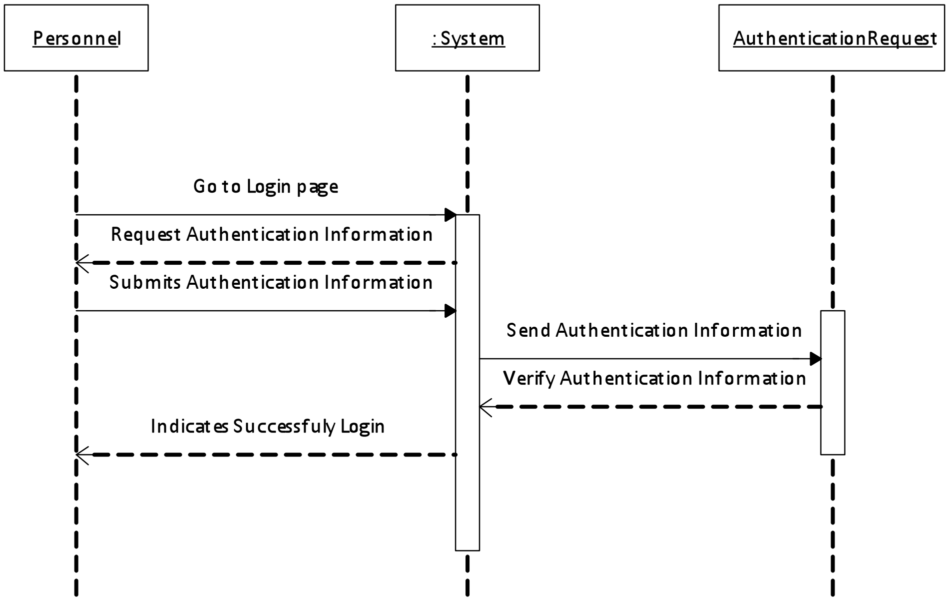

Personnel
: System
AuthenticationRequest
Go to Login page
Request Authentication Information
Submits Authentication Information
Send Authentication Information
Verify Authentication Information
Indicates Successfuly Login

Figure 4.1: Authenticate User system sequence diagram of the main success scenario.

Place Order

The following diagram exhibits the main success scenario for placing orders into at the restaurant. The mobile application will be implementing this method for the waiters. The featured menu will displayed for the user, in this case the waiters, to select from the available foods, beverages, and other items to order. Once the selections are done, the order is submitted to the chef's interface to be prepared. Alternate scenarios will have a similar sequence of events as below. However, instead of the indication of success returned from the system, a message indicating the failed attempt to place an order will be returned.

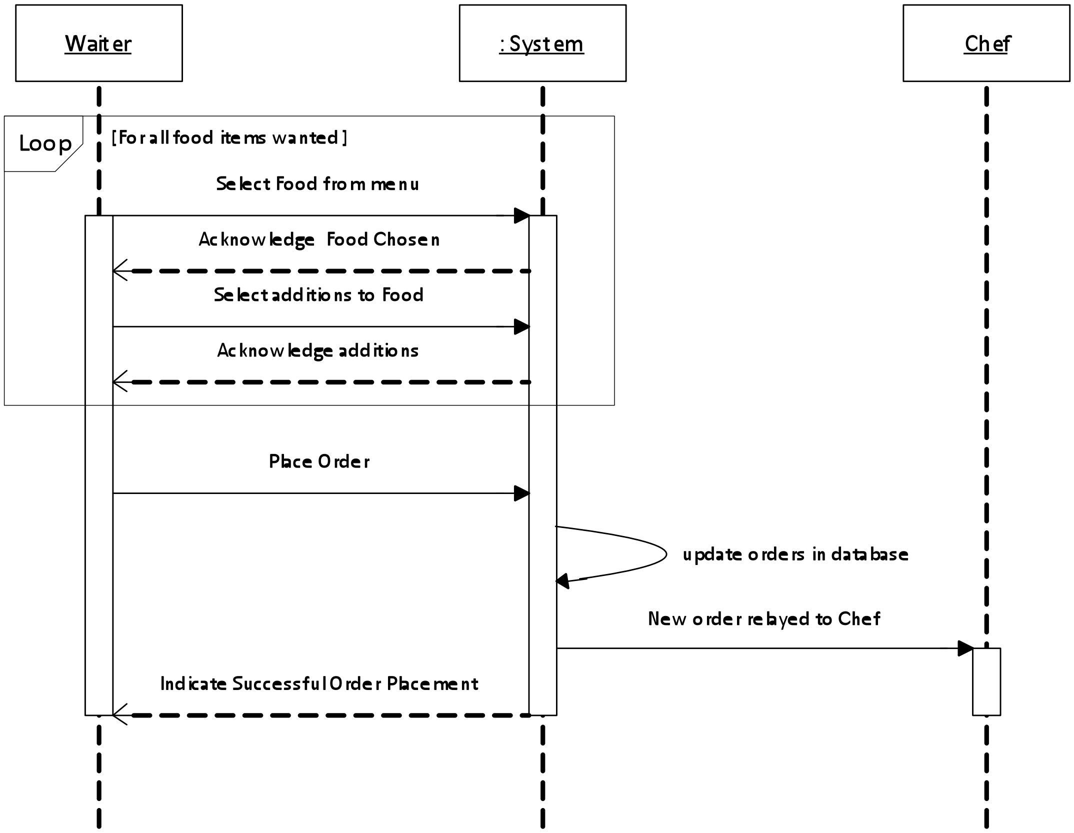


Figure 4.2: System sequence diagram for successfully placing an order(s).

Edit Menu Items

The following diagram shows the possible flow of successful events when editing the menu, which involves the addition and deletion of menu items along with editing existing menu items. Only the manager is given the privileges to make changes to the menu by entering the “EditMenuItems” interface. From here, the current menu list is displayed to allow for the deletion or editing of existing items when selected. To add items, new information must be typed into the provided text fields. The alternate scenarios will be similar to the diagram below with the system returning an error rather than an updated menu.

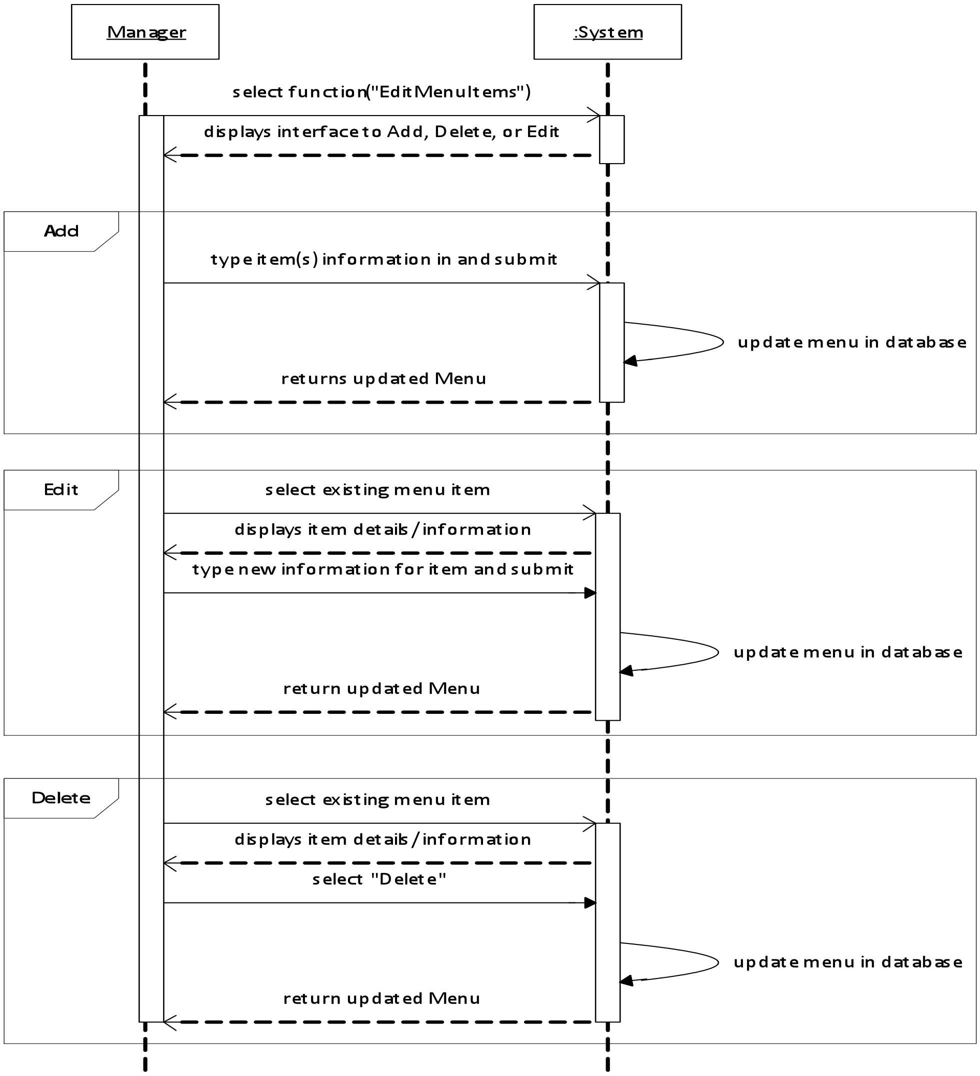

Manager
:System
select function("EditMenuItems")
displays interface to Add, Delete, or Edit
Add
type item(s) information in and submit
update menu in database
returns updated Menu
Edit
select existing menu item
displays item details/information
type new information for item and submit
update menu in database
return updated Menu
Delete
select existing menu item
displays item details/information
select "Delete"
update menu in database
return updated Menu

Figure 4.3: System Sequence diagram for changing the menu.

Edit Employees

The following system sequence diagram shows the series successful of events that can be taken when using editing the employees. This can only be done by the manager, so the diagram shows only the actor Manager with the system. Here, the manager is allowed to edit the employees with updated information including wages, remove any existing employees that are fired or laid off, and add new employees to that have just been hired. For the alternate scenarios, the system will either not allow changes to be made or return an error message to the manager.

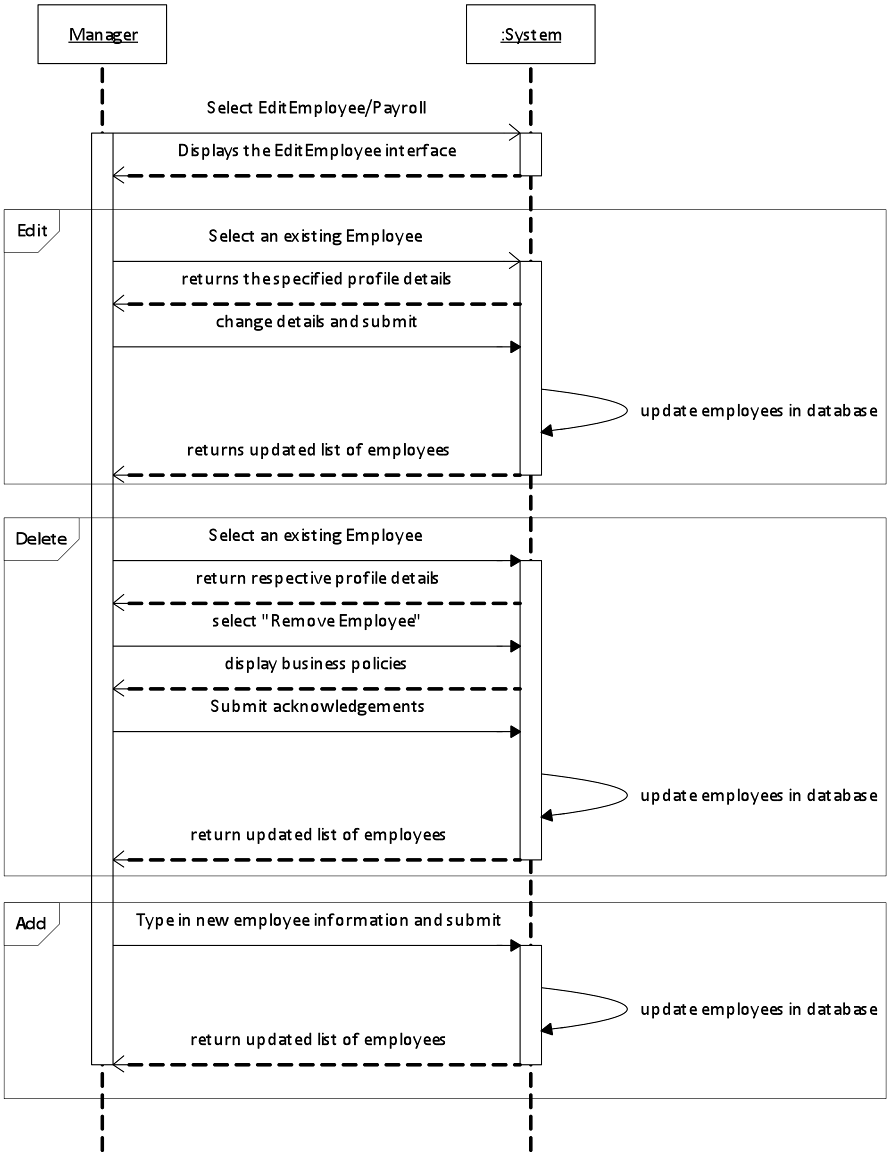

Manager
:System
Select EditEmployee/Payroll
Displays the EditEmployee interface
Edit
Select an existing Employee
returns the specified profile details
change details and submit
update employees in database
returns updated list of employees
Delete
Select an existing Employee
return respective profile details
select "Remove Employee"
display business policies
Submit acknowledgements
update employees in database
return updated list of employees
Add
Type in new employee information and submit
update employees in database
return updated list of employees

Figure 4.4: EditEmployee system sequence diagram of successful events.

Edit Inventory/Stock

The following system sequence diagram shows the sequence of events on the restaurant's inventory, only allowed by the manager. The only possible exception to privileges is the automatic reduction in stock item quantities when orders are cooked. The manager is allowed to add and remove stock items as well as change the current quantities available.

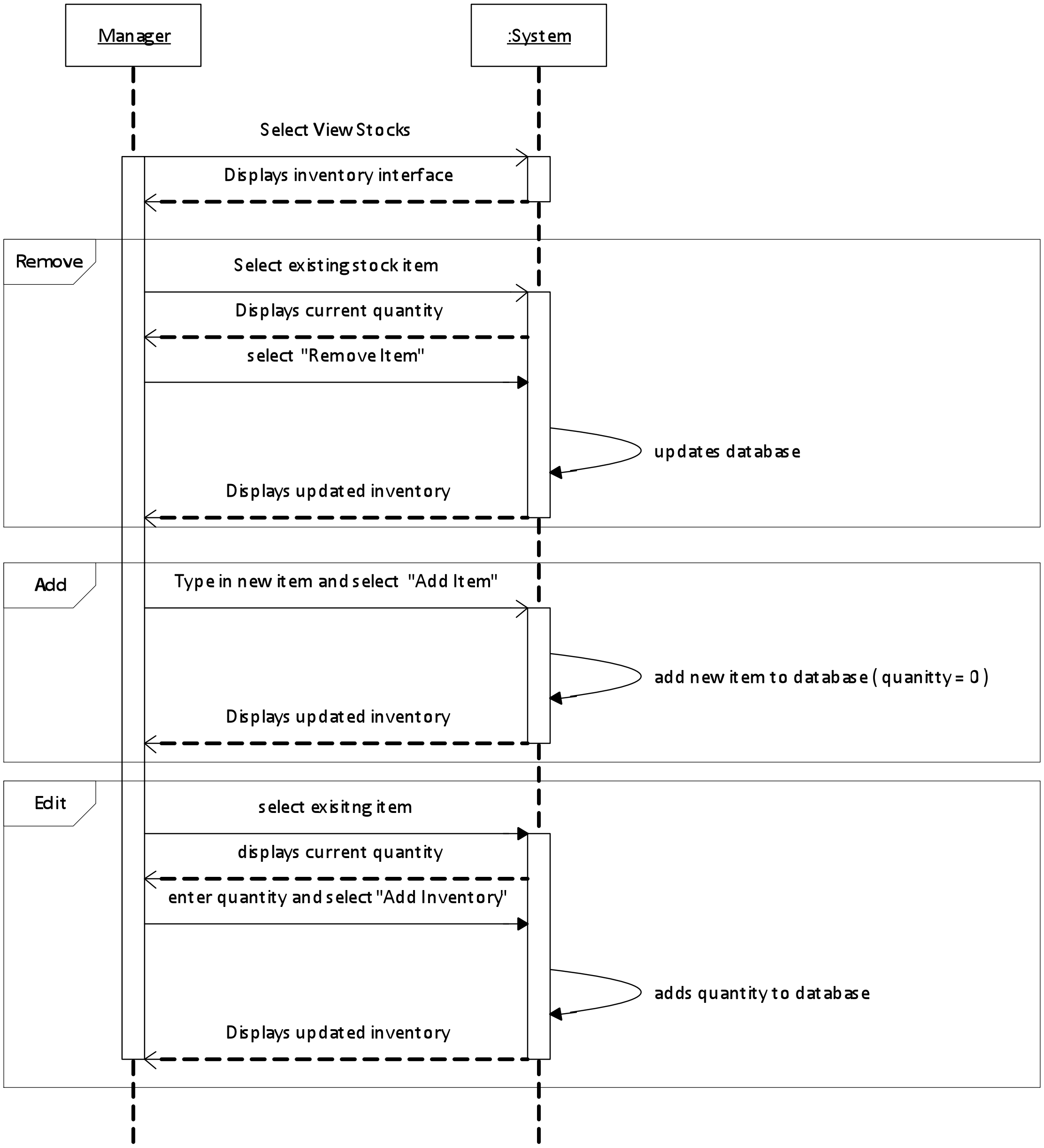

Manager
:System
Select View Stocks
Displays inventory interface
Remove
Select existing stock item
Displays current quantity
select "Remove Item"
updates database
Displays updated inventory
Add
Type in new item and select "Add Item"
add new item to database (quanitty = 0)
Displays updated inventory
Edit
select exisitng item
displays current quantity
enter quantity and select "Add Inventory"
adds quantity to database
Displays updated inventory

Figure 4.5: EditInventory diagram of system sequences.

Clock In / Clock Out

The following system sequence diagram describes the actions taken during a successful scenario of events. All restaurant personnel need to clock in and clock out to record into the system how long they have worked. The difference between the clock in and clock out times are then saved to the database as total time worked.

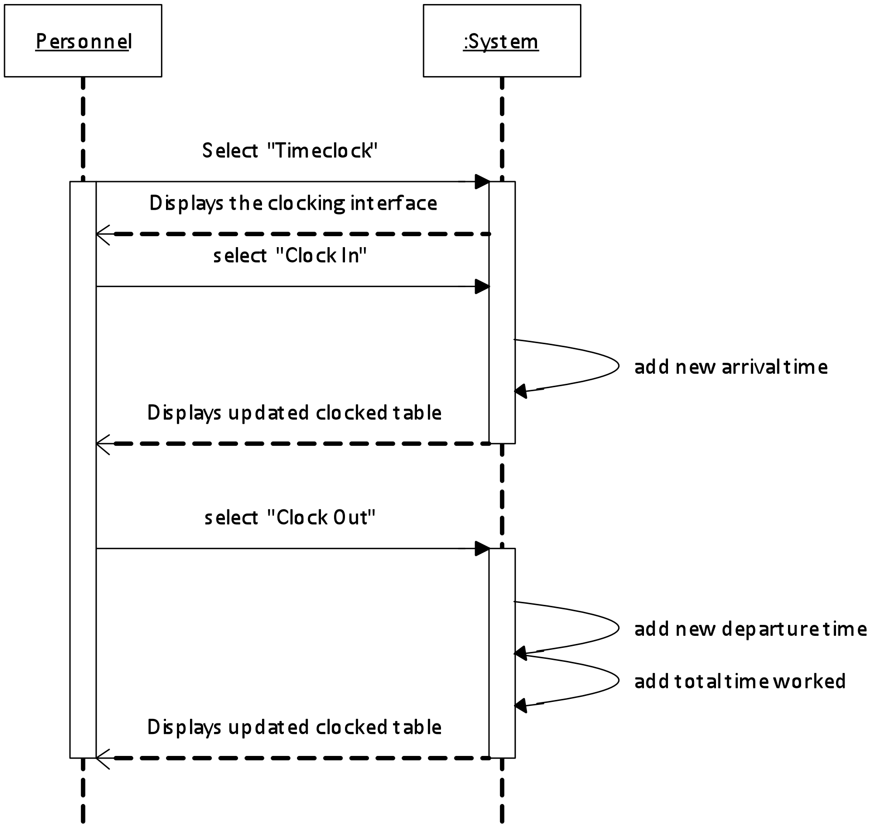


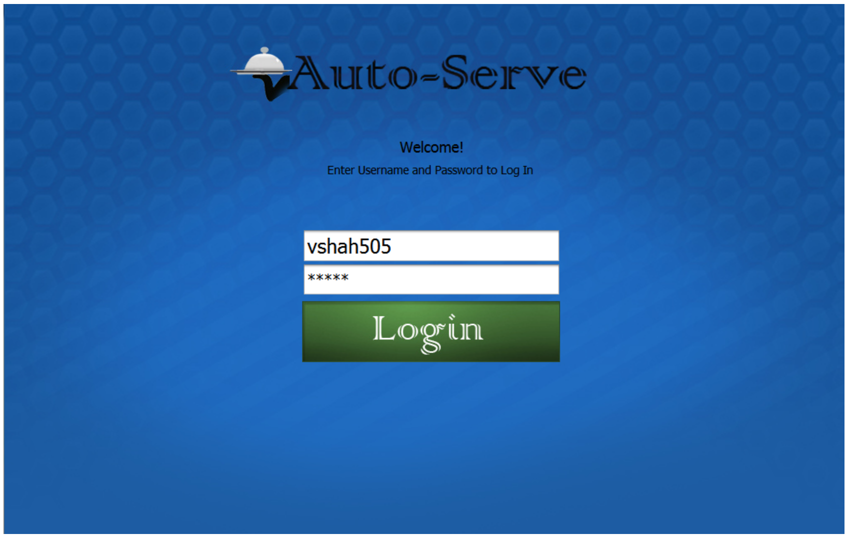
Figure 4.6: Diagram of system sequences involving hours clocked.

IV. USER INTERFACE SPECIFICATION

A. DESIGN

Our interface is specifically designed to be user friendly. Firstly, we tried to create an aesthetically pleasing environment. This should create a feeling that doing their job is more enjoyable.We specifically made the interface have a dominant blue color. According to physiological studies about color, "Seeing the color blue actually causes the body to release chemicals that produce a calming effect...People tend to be more productive when around a blue environment because they are more calm and focused at the task at hand." This is obviously something that a restaurant, an environment that can at times be very busy and stressful, would want. Also, we have implemented big buttons with clear icons and labels. This feature makes it easy for employees to use the touch screen in a expedient manner. Our user interface is also designed to have an intuitive flow, which allows for an almost zero learning curve for a new hire. The desktop interface is as follows.

When one opens our system they are met with a welcome screen that prompts them to log in.



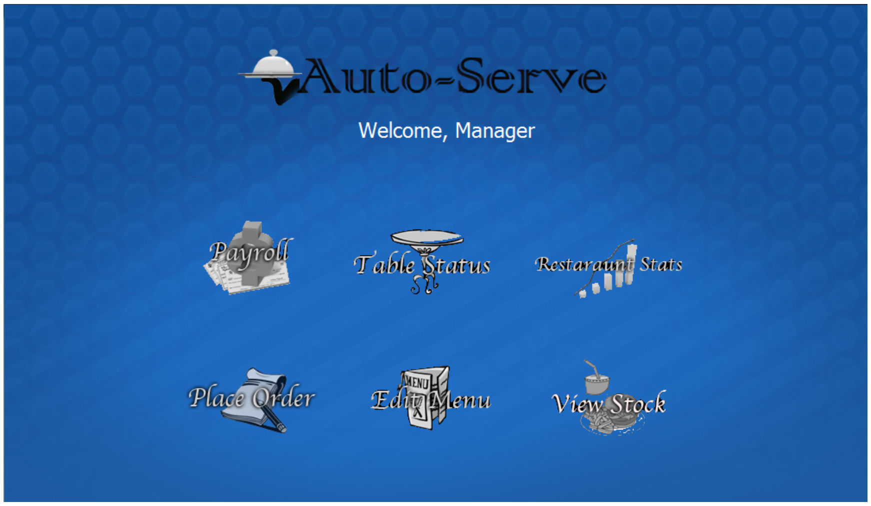
Welcome!

Enter Username and Password to Log In

vshah505

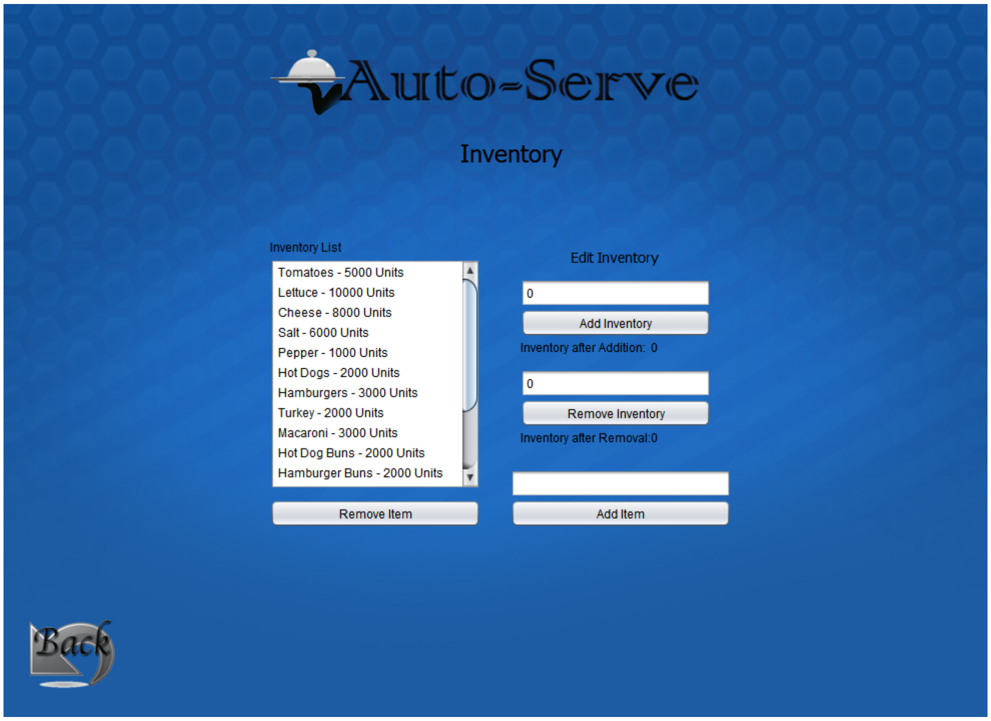
*****

This Log-in procedure not only allows the system to verify who is using the system, but also allows it to know what system feature privileges to give to the user. For examples, when a manager logs in, our system will know it is a manager and thus display the manager options screen.



Welcome, Manager

The "View Stock" button allows the manager to view the status of all the restaurants ingredients(low stock, high stock, etc).The "Restaurant Stats" button shows the manager various restaurant statistics such as what items are selling the best, business reports, and more.



Auto-Serve

Inventory

Inventory List

Tomatoes - 5000 Units
Lettuce - 10000 Units
Cheese - 8000 Units
Salt - 6000 Units
Pepper - 1000 Units
Hot Dogs - 2000 Units
Hamburgers - 3000 Units
Turkey - 2000 Units
Macaroni - 3000 Units
Hot Dog Buns - 2000 Units
Hamburger Buns - 2000 Units

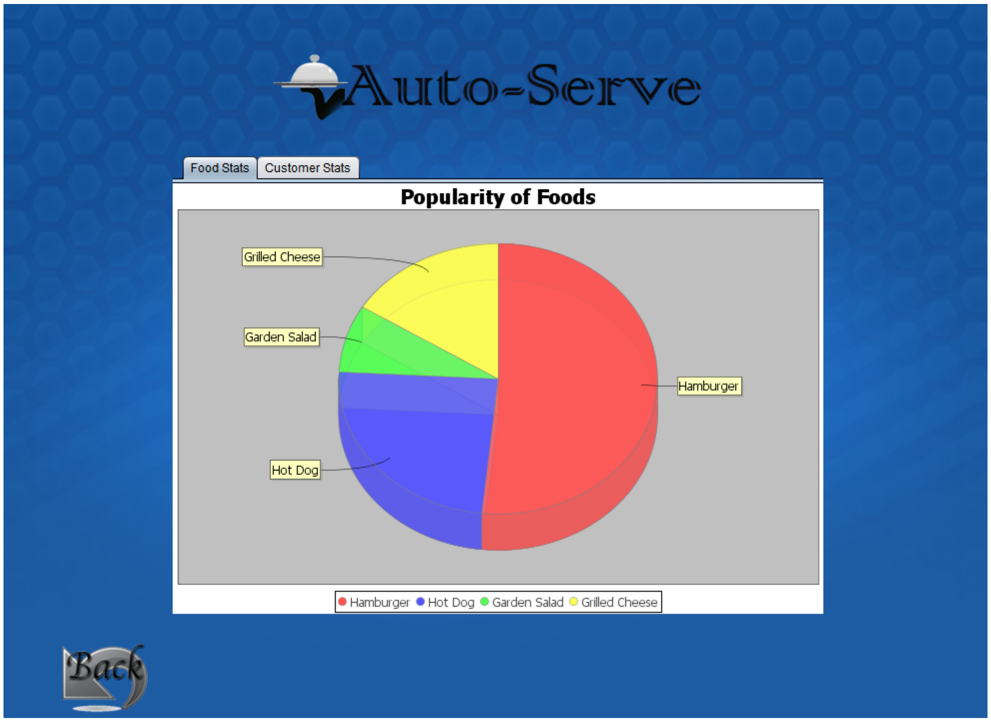
Edit Inventory

0

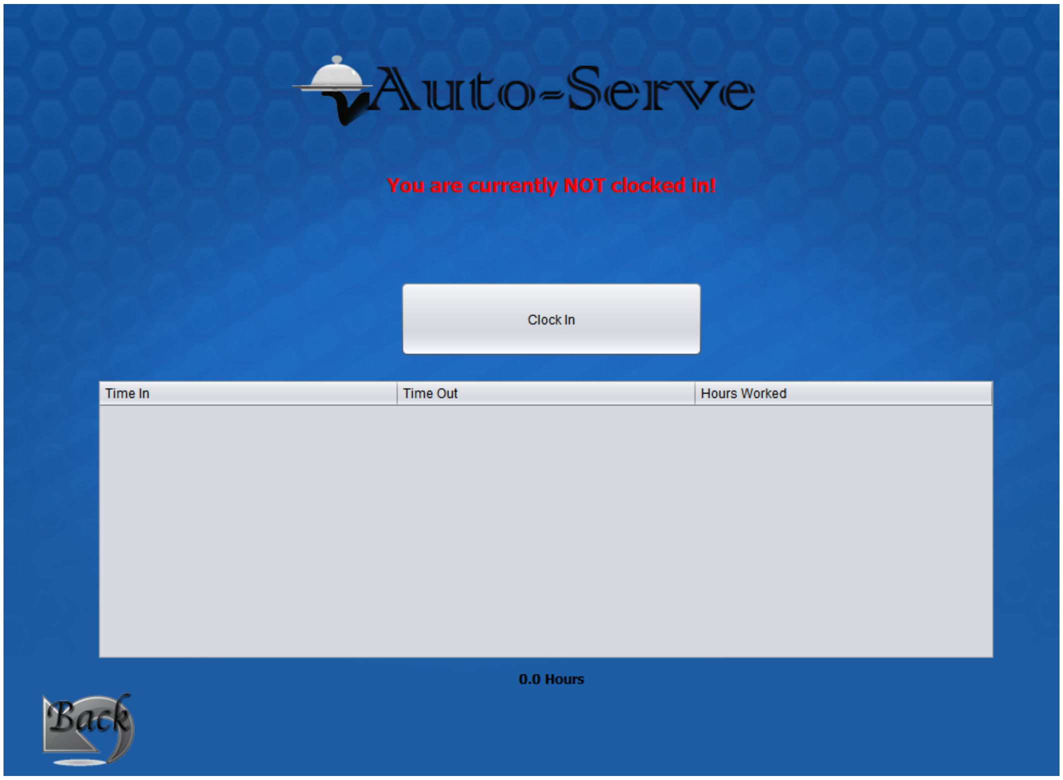
Inventory after Addition: 0

0

Inventory after Removal:0



The “Timeclock” button lets the user clock in or out of their shift.

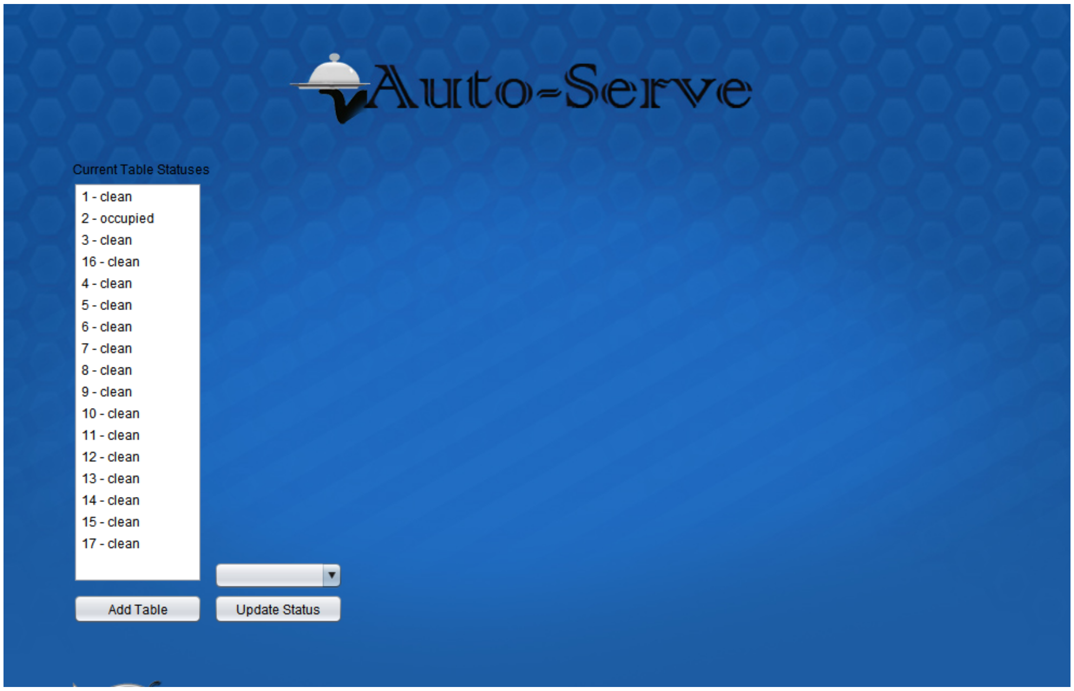


You are currently NOT clocked in!

Time In	Time Out	Hours Worked

0.0 Hours

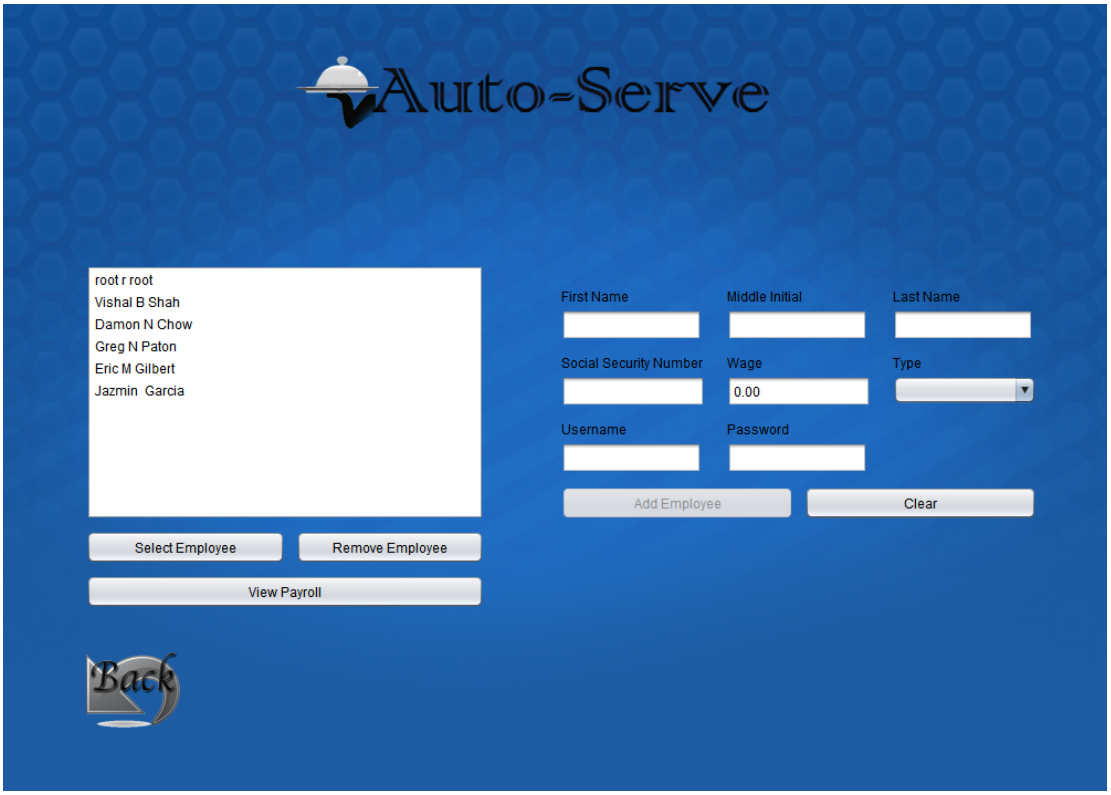
The “Table Status” button lets the user view and edit the status of a table; “Dirty”, “Occupied”, or “Clean”.



Current Table Statuses

1 - clean
2 - occupied
3 - clean
16 - clean
4 - clean
5 - clean
6 - clean
7 - clean
8 - clean
9 - clean
10 - clean
11 - clean
12 - clean
13 - clean
14 - clean
15 - clean
17 - clean

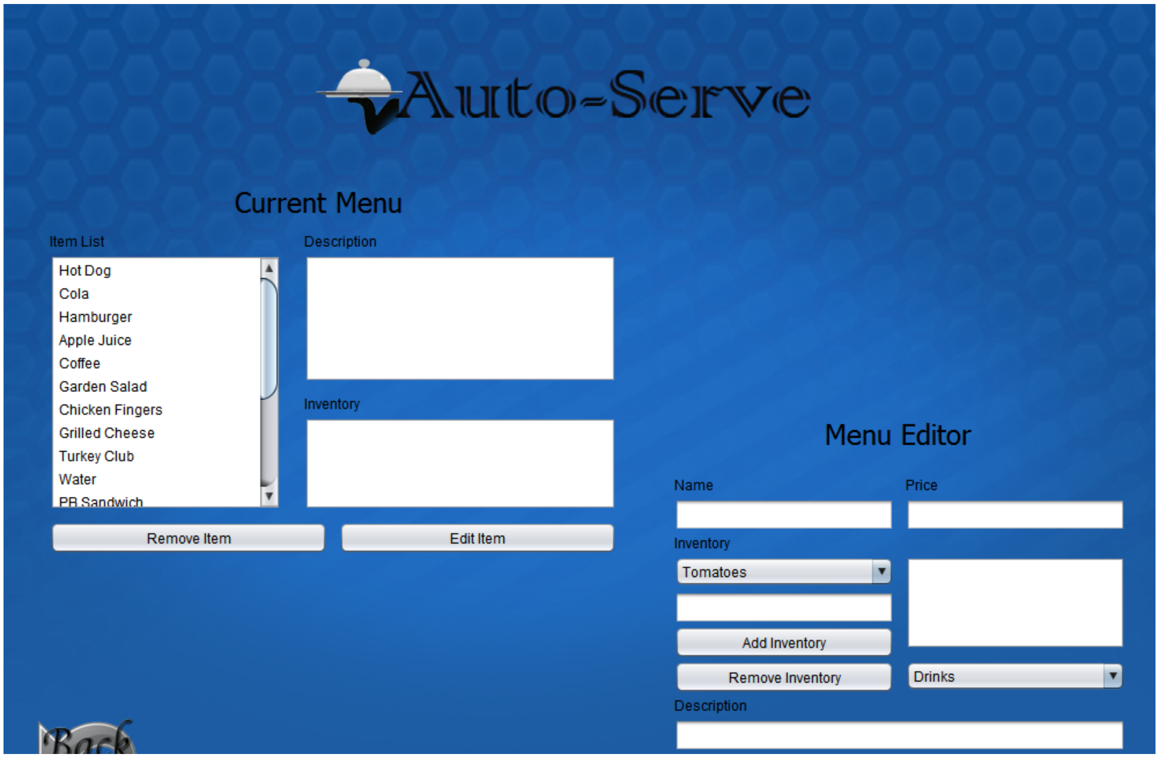
The “Payroll” button allows the manager to add and delete employees, as well as alter employee time cards in case of a mistake.



root r root
Vishal B Shah
Damon N Chow
Greg N Paton
Eric M Gilbert
Jazmin Garcia

Wage: 0.00

The “Edit Menu” button lets the manager make changes to the restaurant menu.

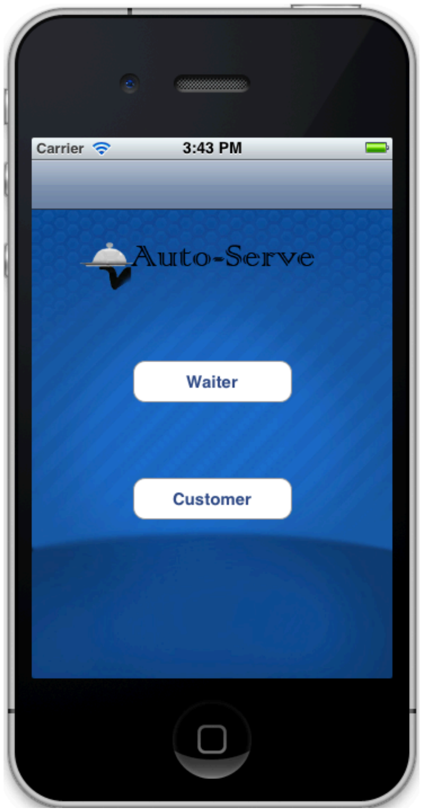


Hot Dog
Cola
Hamburger
Apple Juice
Coffee
Garden Salad
Chicken Fingers
Grilled Cheese
Turkey Club
Water
PB Sandwich

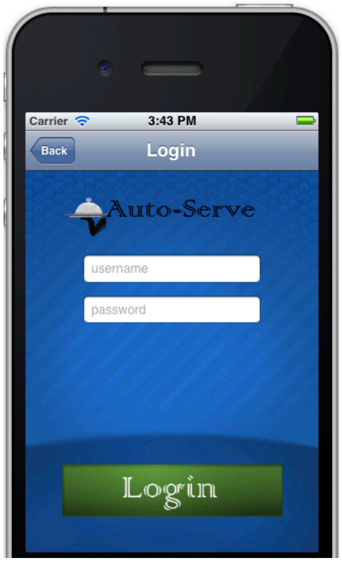
Inventory: Tomatoes

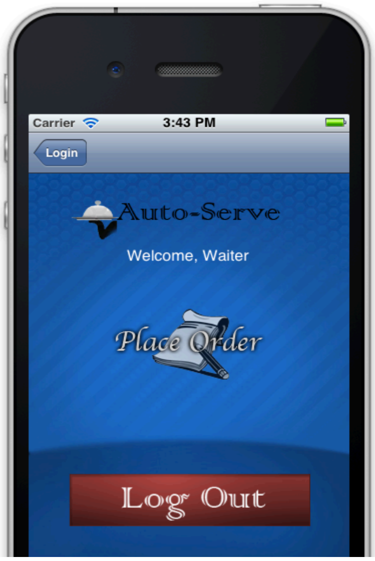
Drinks

To compliment the desktop interface, we created a mobile interface to be used for actually placing orders either by the waiter or the customers themselves if they choose to do so. When the mobile app is first launched, you are shown "User-Mode" screen.

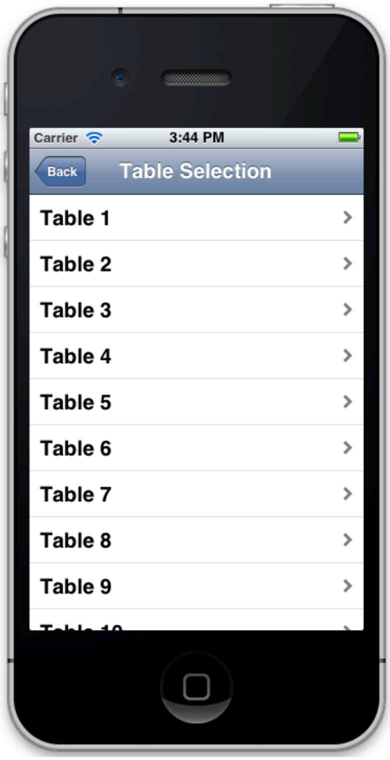


If a waiter is taking orders for a table, they would choose the "Waiter" option. If the restaurant is employing embedded devices for the customer to take the order themselves, they would choose "Customer". This then takes you to the Log-in screen and then the prompt to start taking orders.

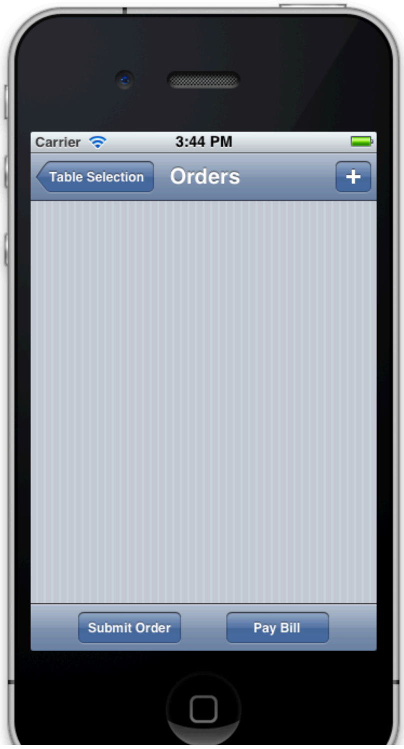




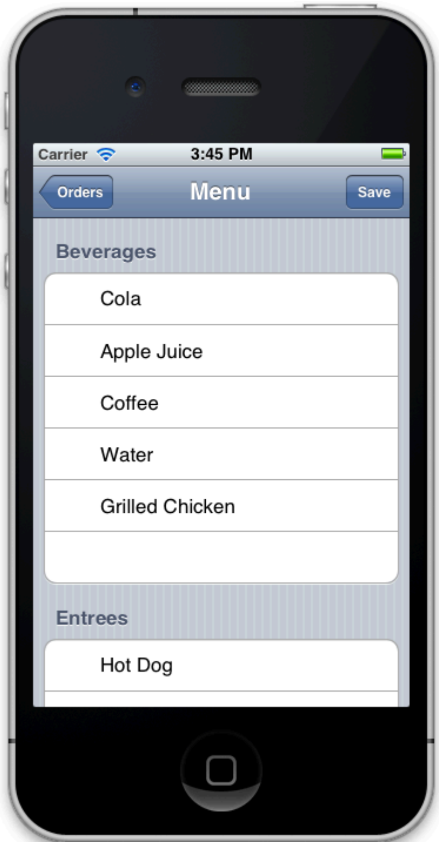
After the waiter presses the “Place Order” button, they must choose which table they are taking orders for.



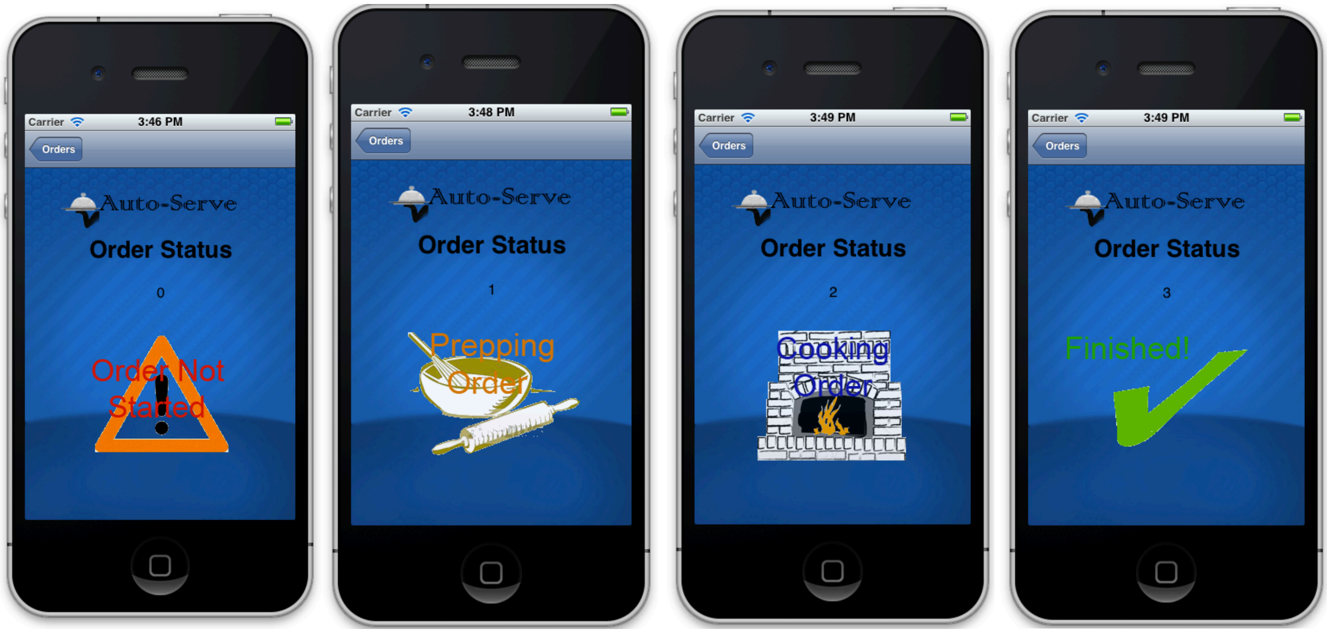
Once the waiter chooses the table, they are met with the “Orders” screen. This screen will show the waiter what the current order is for that table, as well as the option to “Sumbit” the order and “Pay Bill”.



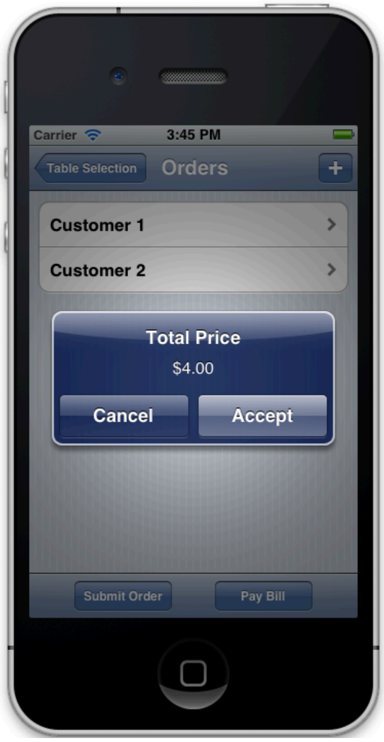
The waiter then presses the "+" button to show the menu and start taking orders for each individual customer at the table. This menu is dynamic and can instantly be altered if the manager makes changes on the desktop application.



If in the "Customer Mode", after they submit their order they are shown the "Order Status" screen. This screen shows progress of their order set by the Chef.



If in “Waiter Mode” after the order is submitted, cooked, and eaten, the waiter can then hit the “Pay Bill” button. This button does a few different things. First, it shows the total price for the tables order. Once the waiter hits “Accept” it will then set that tables status to “dirty” so the bus boy knows it needs to be cleaned. Lastly, it launches the Square Credit Card Reader application to finalize the transaction. This application allows the restaurant to take payments easily and efficiently. The card reader plugs in to the headphone jack of the iOS device and allows the customer to swipe their credit/debit card to pay for their meal. They can sign their name with their finger and choose to have their receipt either emailed or texted to them.



B. USER EFFORT ESTIMATION

The most used and typical usage scenario is the log in of the restaurant personnel. This requires input of information for authentication by the system. As a result, there is little to no navigational input requirements as seen below.

Logging In (Applies to all restaurant personnel)

1. Navigational: total 1 mouse click as follows [after completing data entry as shown below]
    a. Click "Login" button to log into the system
2. Data Entry: total 2 mouse clicks and a maximum of 50 keystrokes as follows
    a. Click the "Username" text field
    b. Enter a user name for the restaurant personnel
    c. Click the "Password" text field
    d. Enter a corresponding password to the entered user name

Customer

The customer user interface (UI) will include 3 prominant options in the main menu for ease-of-use and reduced complexity. This includes a "Menu" button for ordering food, a "Request Check" button to ask for a check and receipt once dining is finished, and a "Call Waiter" button for further assistance or beckoning a waiter to order for the table instead. A "Home" button will bring the customer back to the main menu after visiting 1 of the three menus.

As a result, this user interface is more demanding of clerical data entry for menu orders as shown below.

<u>Placing an Order</u>

1. Navigation: total 3 mouse clicks as follows
    a. Click "Menu" button [after completing data entry as shown below]
    b. Click "Submit Order" button to place order
2. Data Entry: total 3 mouse clicks as follows
    a. Click drop-down menu under order wanted
    b. Click option from drop-down menu
    c. Click "Add to Order" button

Chef

The user effort for the chefs' user interface will be simplified for quick and efficient use in the kitchen setting. The buttons for progress are made large and made to appear in sequence with only one progress status update on the screen at a time. This allows for more concentration and time for cooking an order well than spending time updating the progress of cooking.

Given this layout, there is little navigational input needed compared to clerical data entry for updating the food completion status.

Updating Order Progress

1. Navigation: total 1 mouse click as follows
    a. Click “Order Progress” button [after completing data entry as shown below]
    b. Click “Exit” button to finish
2. Data Entry: total 5 mouse clicks as follows
    a. Click drop-down menu for table orders
    b. Click order to edit from drop-down menu
    c. Click “Prepping” button
    d. Click “Cooking” button
    e. Click “Order Ready” button

Busboy

The busboy mainly needs to check the table status. Therefore, the prominent functionality allows the busboy to know which tables need cleaning as efficiently as possible.

The navigational input required is minimized here, with more clerical data entry for changing table statuses.

Viewing Table Status / Cleaning Tables

1. Navigation: total 2 mouse clicks as follows
    a. Click “Table Status” button [after completing data entry as shown below --
    b. Click “Exit” button to return to main menu
2. Data Entry: total 3 mouse clicks as follows [after cleaning a dirty table]
    a. Click “Edit” tab
    b. Click on the table just cleaned
    c. Click “Clean” to change status to clean

Hostess

Aside from checking the table status to know which table to direct to customers to, the hostess will also be allowed to make reservations. As a result, most of the inputs will be within the clerical data entry.

A majority of the inputs for the hostess are for recording a reservation into the system, which is demands more clerical data entry than navigation.

Make a Reservation

1. Navigation: total 2 mouse clicks as follows
    a. Click “Reservations” button [after completing data entry as shown below]
    b. Click “Create Reservation” to make a reservation

2. Data Entry: total 7 mouse clicks and a maximum of 25 keystrokes as follows
    a. Click the "Customer Name" text field
    b. Enter customer's name
    c. Click the "Date" drop-down menu
    d. Click the specified date of the reservation
    e. Click the "Time" drop-down menu
    f. Click the time of day the reservation was made
    g. Click the "Party Size" drop-down menu
    h. Click the quantity that applies

Manager

Since the manager overlooks most, if not all of the restaurant operations, he/she will have the greatest number of menu options. There is a greater navigational effort for the manager to view restaurant statistics and operations. Some functionalities will have more emphasis on data entry, including changes to menu options and employee/payroll statuses.

The following only requires navigational input, which has been minimized for ease-of-access.

Viewing Popularity of Menu Options

1. Navigation: total 3 mouse clicks as follows
    a. Click "Restaurant Stats" button
    b. Click "Item History" Tab
    c. Click "Exit" button to return to main menu
2. Data Entry: no data entry required

C. EFFORT ESTIMATION USING USE CASE POINTS

Using the formulas from the notes on Use Case Estimation, we formulate the following calculations.

Actors	Weight
Waiter	2
Hostess	2
Manager	3
Chef	1
Busboy	1

The number 1 is associated with Simple, 2 is associated with average, and the 3 is associated with Complex. Below is the UAW calculation which is the unadjusted actor weight.

UAW - (1*2)+(2*2)+(3*1)=9

For the use cases, we find the below calculations.

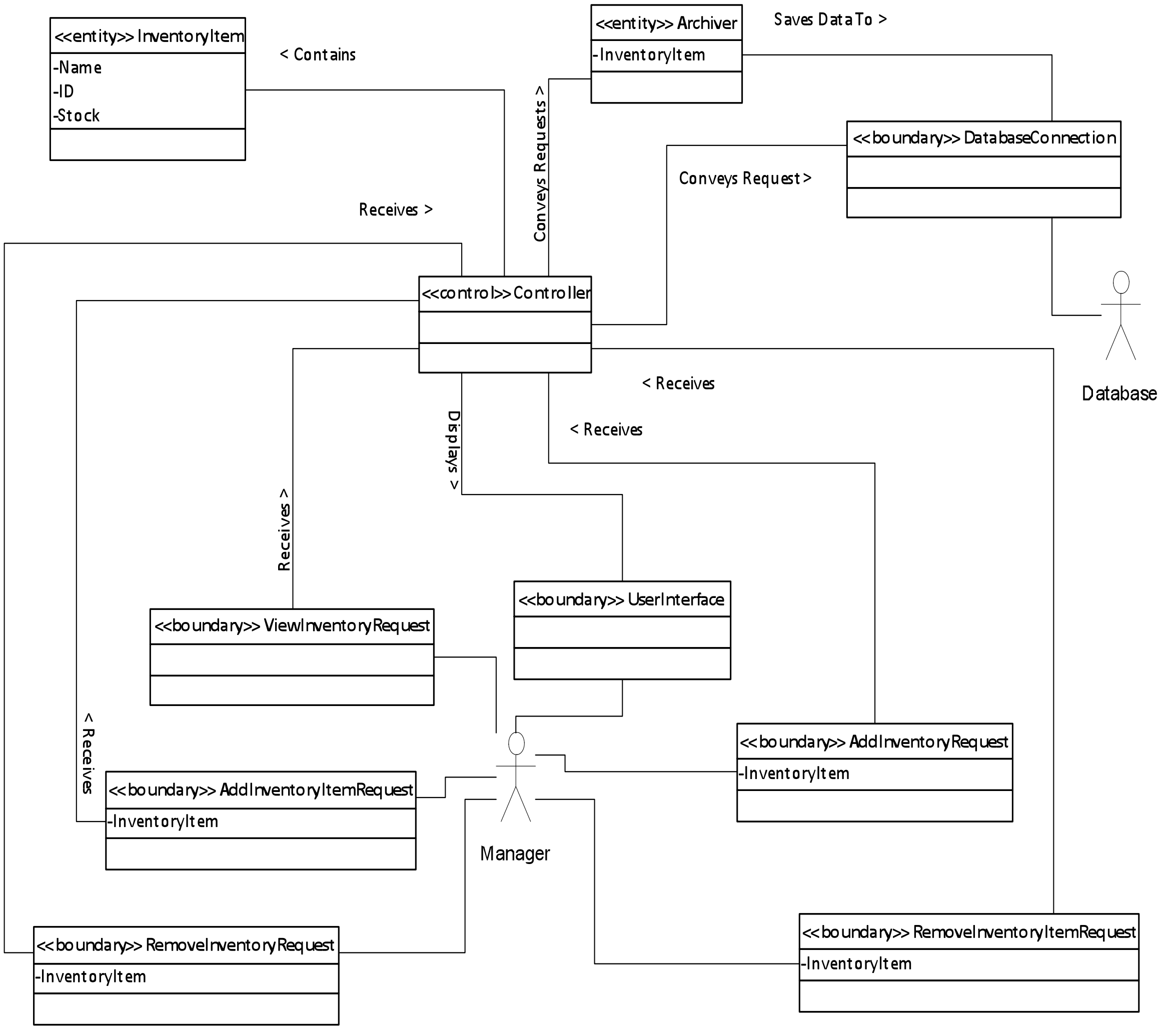
Use Cases	Weights
ViewStockStats	2
PlaceOrder	2
CheckTableStatus	1
PayBill	1

UUCW = (1*1)+(5*2) + (1*2) = 13

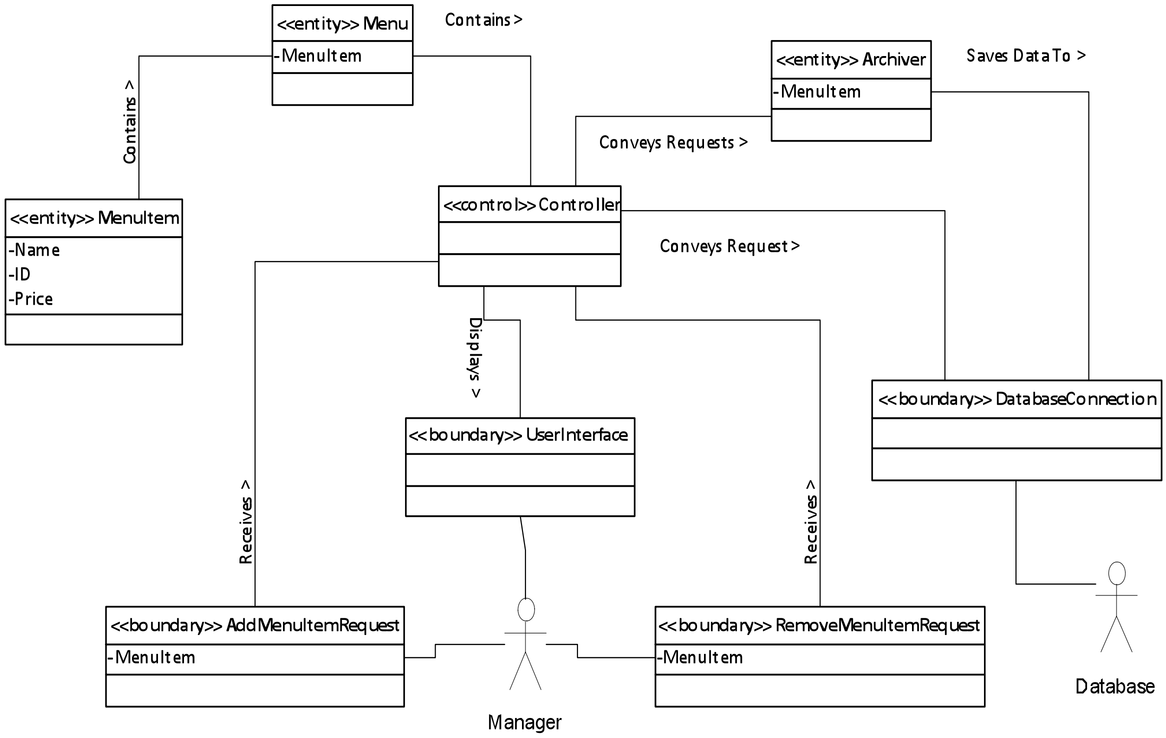
V. DOMAIN ANALYSIS

A. DOMAIN MODEL

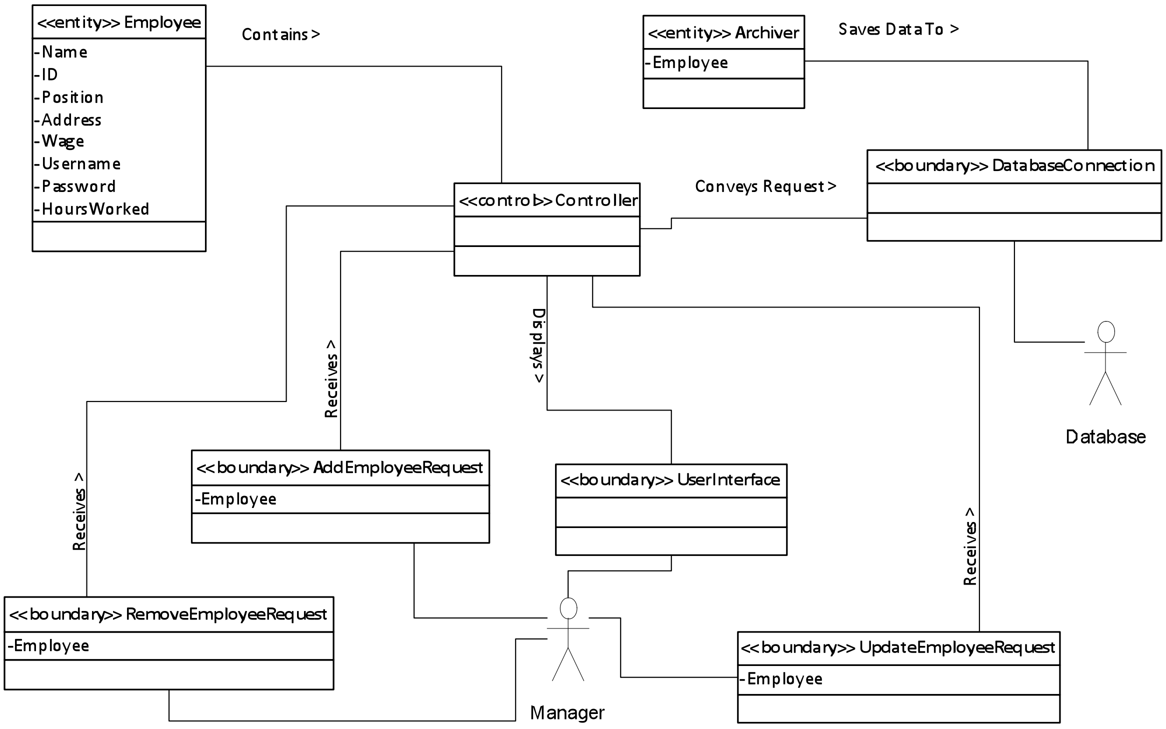
Inventory Domain



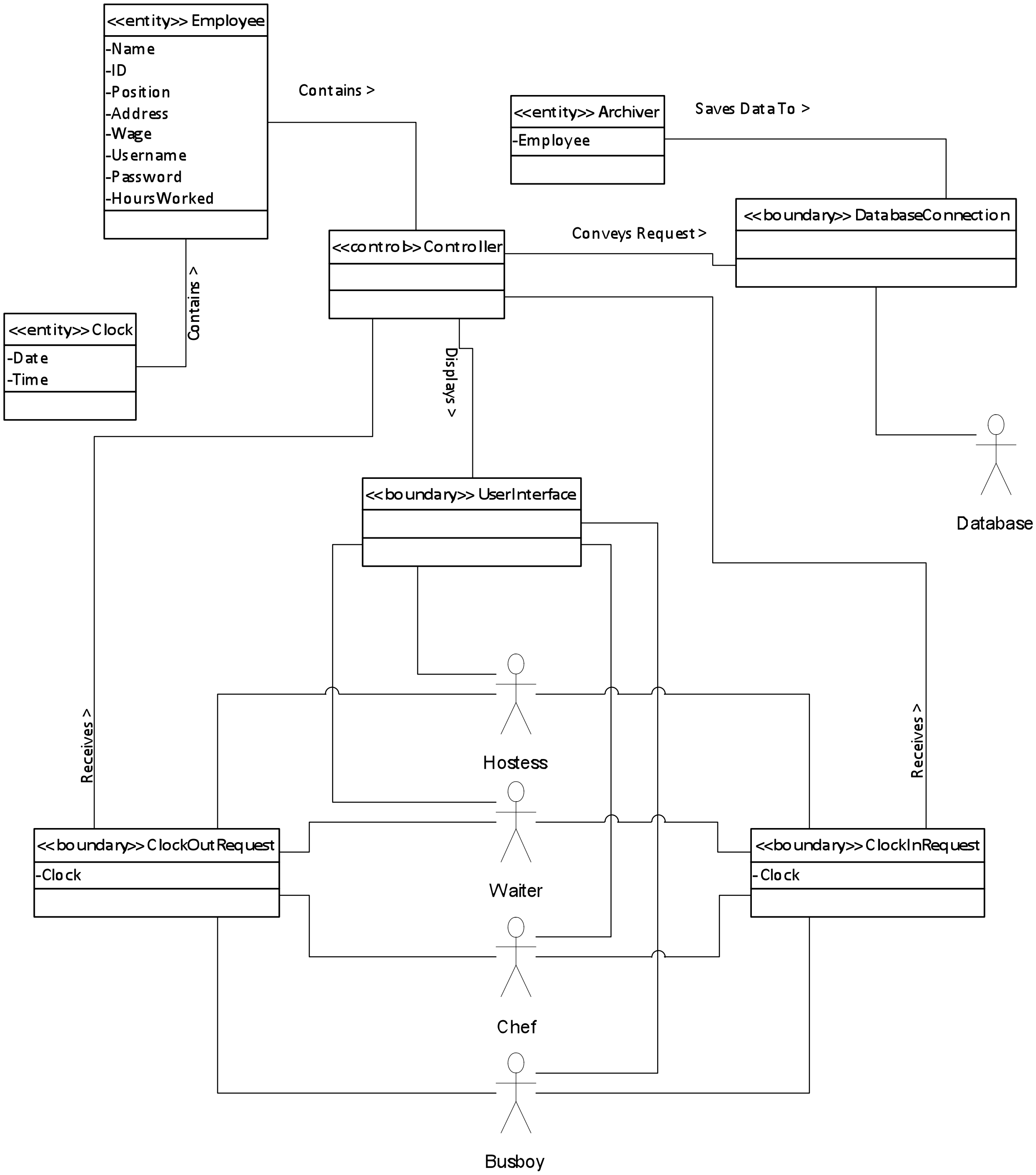
Menu Domain



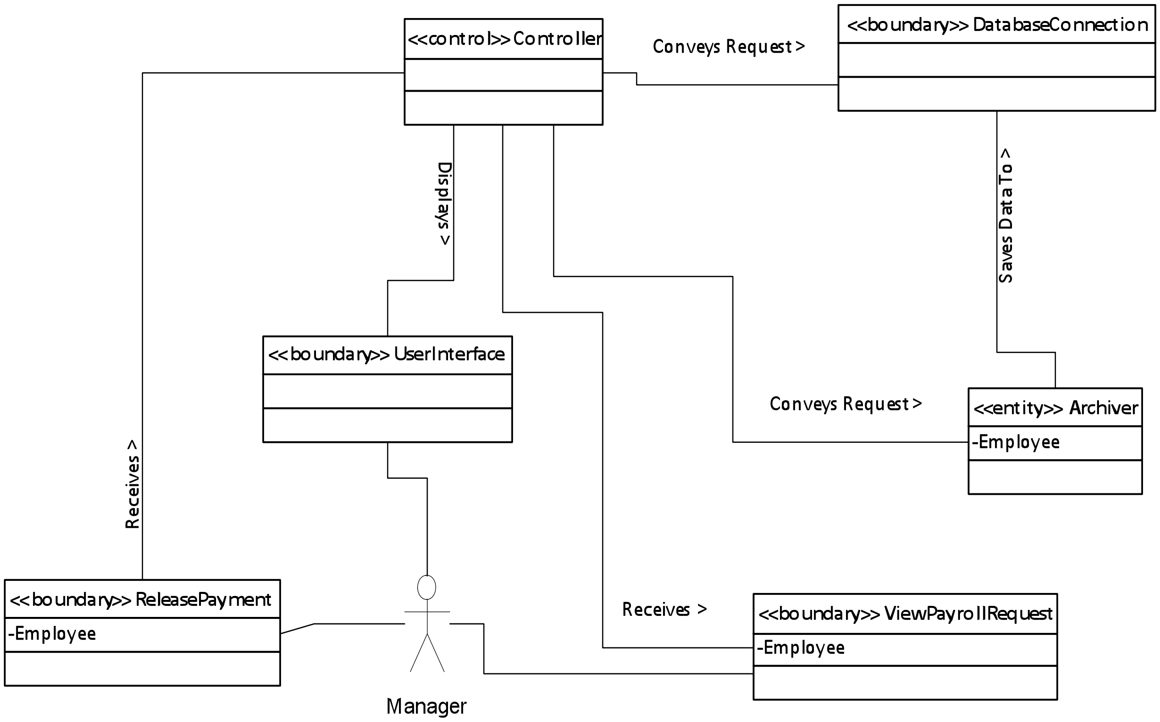
Employee Domain



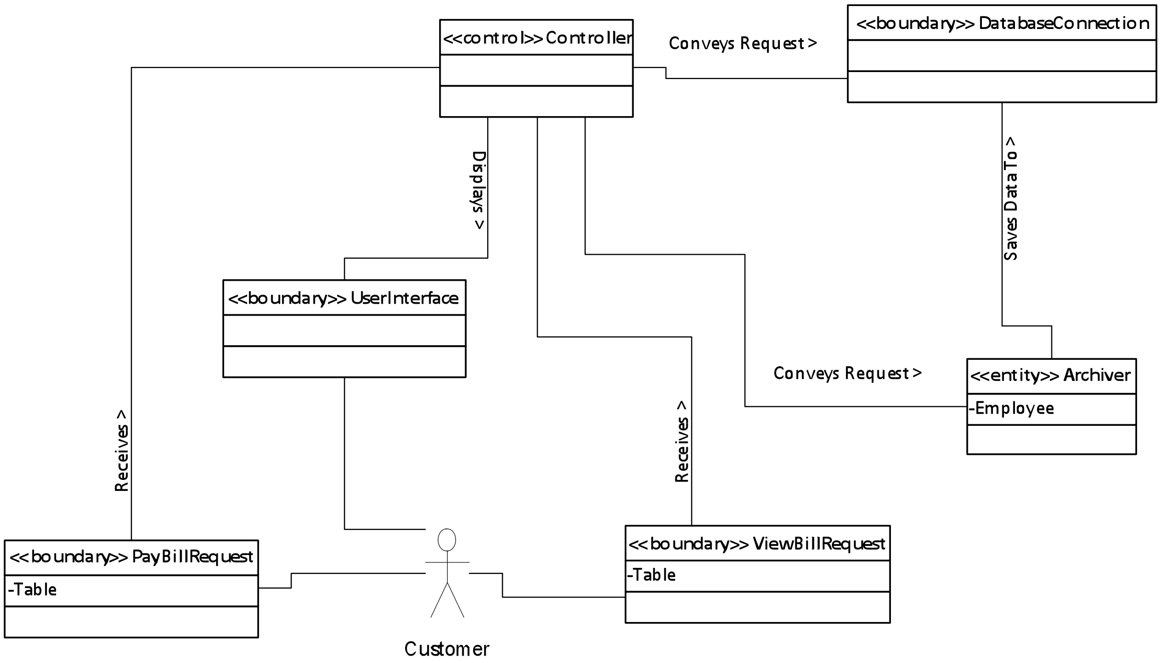
Clocking Domain



Payroll Domain



PayBill Domain



All of the domain models are linked by the controller, which is the main hub of the program's operation.

I. CONCEPT DEFINITIONS

Responsibility Description	Type	Concept Name
Container holding FoodItems that customers order.	K	Order
Coordinate actions of the ordering sub-system and delegate the work to other concepts.	D	OrderController
Archive a request in the database in the appropriate table.	D	OrderArchiver
Form specifying the order a customer places that is placed in the database.	D	OrderRequest
Form specifying a status update of an order that is placed in the database.	D	FoodProgressUpdateRequest
Returns the current status of a specified order that is present in the database.	D	OrderStatusRequest
Archive a request in the database in the appropriate table.	D	EmployeeArchiver
Verify user credentials that were entered with those in the profile of a specific employee.	D	AuthenticateUser
Coordinate actions of the employee sub-system and delegate the work to other concepts.	D	EmployeeController
Form specifying changes to an employee profile that is updated in the database.	D	EditEmployeeRequest
Container holding contact and payroll information about employees.	K	EmployeeProfile
Container holding information about food that can be served.	K	FoodItem
Container holding a list of FoodItems.	K	Menu
Coordinate actions of the menu sub-system and delegate the work to other concepts.	D	MenuController
Archive a request in the database in the appropriate table.	D	MenuArchiver
Form specifying changes to the list of items and the items that is updated and saved in the database.	D	EditMenuRequest
Container holding information about food items in inventory.	D	InventoryItem
Archive a request in the database in the appropriate table.	D	InventoryArchiver
Coordinate actions of the inventory sub-system and delegate the work to other concepts.	D	InventoryController
Form specifying changes to inventory that are saved in the database.	D	EditInventoryRequest
Container holding information about the tables in the establishment.	K	Table
Archive a request in the database in the appropriate table.	D	TableArchiver
Coordinate actions of the table sub-system and delegate the work to other concepts.	D	TableController
Returns the current status of a specified table that is present in the database.	D	CheckTableStatusRequest
Form specifying changes to the status of a specified table that is currently in the database.	D	EditTableStatusRequest
Form specifying changes to the physical layout of the tables in	D	EditTableLayoutRequest

the establishment.		
Form specifying log in credentials that is later verified.	D	AuthenticationRequest
Form that shows actor current context, the actions that can be done, and the outcomes of previous actions.	K	InterfacePage
Coordinate actions of the entire system and delegate the work to other concepts.	D	MainController
Form specifying modification and calculation of payroll.	D	PayrollRequest
Form specifying a reservation for a certain date and time.	D	ReservationRequest

II. ASSOCIATION DEFINITIONS

Concept Pair	**Association Description**	**Association Name**
OrderController <-> OrderArchiver	OrderController conveys request to OrderArchiver to send data to the database.	conveys request
OrderController <-> OrderRequest	OrderController receives order from OrderRequest and sends order to OrderArchiver	receives
OrderController <-> OrderStatusRequest	OrderController receives request from OrderStatusRequest and returns status of order from databse.	receives
OrderController <-> FoodProgressUpdateRequest	OrderController receives request from FoodProgressUpdateRequest and sends the modified data to OrderArchiver.	receives
OrderController <-> DatabaseConnection	OrderController conveys request to DatabaseConnection and receives requested data.	conveys request
OrderController <-> MainController	MainController conveys request to OrderController, which delegates task to other concepts.	conveys request
EmployeeController <-> PayrollRequest	EmployeeController receives a request from PayrollRequest, which is then processed and sent to the database.	receives
EmployeeController <-> EditEmployeeRequest	EmployeeController receives a request from EditEmployeeRequest and sends data to EmployeeArchiver.	receives
EmployeeController <-> AuthenticateUser	EmployeeController conveys request to AuthenticationUser and receives validity of authentication.	conveys request
EmployeeController<->	EmployeeController conveys request	conveys request

EmployeeArchiver	to EmployeeArchiver to send data to the database.	
EmployeeController <-> DatabaseConnection	EmployeeController conveys request to DatabaseConnection and receives requested data.	conveys request
EmployeeController <-> MainController	MainController conveys request to EmployeeController, which delegates task to other concepts.	conveys request
TableController <-> EditTableLayoutRequest	TableController receives request from EditTableLayoutRequest and sends data to TableArchiver	receives
TableController <-> EditTableStatusRequest	TableController receives request from EditTableStatusRequest and sends data to TableArchiver	receives
TableController <-> CheckTableStatusRequest	TableController receives request from CheckTableStatusRequest and returns status.	receives
TableController <-> TableArchiver	TableController conveys request to TableArchiver to send data to the database.	conveys request
TableController <-> DatabaseConnection	TableController conveys request to DatabaseConnection and receives requested data.	conveys request
TableController <-> MainController	MainController conveys request to TableController, which delegates task to other concepts.	conveys request
InventoryController <-> EditInventoryRequest	InventoryController receives request from EditInventoryRequest and sends data to InventoryArchiver	receives
InventoryController <-> InventoryArchiver	InventoryController conveys request to InventoryArchiver to send data to the database.	conveys request
InventoryController <-> DatabaseConnection	InventoryController conveys request to DatabaseConnection and receives requested data.	conveys request
InventoryController <-> MainController	MainController conveys request to InventoryController, which delegates task to other concepts.	conveys request
MenuController <-> EditMenuRequest	MenuController receives request from EditMenuRequest and sends data to MenuArchiver.	receives
MenuController <-> MenuArchiver	MenuController conveys request to MenuArchiver to send data to the	conveys request

	database.	
MenuController <-> DatabaseConnection	MenuController conveys request to DatabaseConnection and receives requested data.	conveys request
MenuConroller <-> MainController	MainController conveys request to MenuController, which delegates task to other concepts.	conveys request
MainController <-> InterfacePage	MainController displays InterfacePage based on actor's position.	display
MainController <-> ReservationRequest	MainController receives request from ReservationRequest and returns feedback of reservation.	receives

III. ATTRIBUTE DEFINITIONS

Concept	**Attributes**	**Attribute Description**
EmployeeProfile	Name	Used to determine the name of the employee for convenient search and reference.
	ID	Holds the identification number of the employee to specify a unique employee.
	Position	Used to determine the employee's position in the infrastructure of the establishment and determine the interface the user is authorized to view.
	Address	Used in order to determine the address at which information will be sent to the user.
	Wage	Used to determine the payment that will be made during payroll calculation.
	Username	Unique phrase used by the user as the log in name and half of the authentication protocol.
	Password	Unique phrase used in conjunction with Username to validate user into system.
	HoursWorked	Holds the total time the user is clocked in.
FoodItems	Name	Used to determine the name of an item for convenient reference.
	ID Number	Holds the identification number for an item in order to

		distinguish unique items.
	Price	Used to determine the cost of an order including an item.
	Description	Holds the detailed information about a menu option.
Order	FoodItem	Holds the menu option chosen to be prepared.
InventoryItem	Name	Used to determine the name of an item in stock within the inventory.
	ID Number	Holds the identification number to determine specific items and organizing.
	Quantity	Holds the amount of an item remaining.
	Description	Holds detailed information on an item.
AuthenticationRequest	Username	Unique phrase stored in the system to match with corresponding entered log in information.
	Password	Unique phrase corresponding to a Username stored for matching with entered keystrokes.
ReservationRequest	Date	Holds the month, day, and year.
	Time	Holds the time of day.
Table	ID Number	Holds the identification number to distinguish between different tables.
	Location	Holds information of the physical placement of tables in the restaurant.

IV. TRACEABILITY MATRIX

Use Case	PW	Order	OrderController	OrderArchiver	OrderRequest	FoodProgressUpdateRequest	OderStatusRequest	EmployeeArchiver	AuthenticateUser	EmployeeController	EditEmployeeRequest	EmployeeProfile	FoodItem	Menu	MenuController	MenuArchiver	EditMenuRequest	InventoryItem	InventoryArchiver	InventoryController	EditInventoryRequest	Table	TableArchiver	TableController	CheckTableStatusRequest	EditTableStatusRequest	EditTableLayoutRequest	AuthenticationRequest	MainController	PayrollRequest	InterfacePage	ReservationRequest
UC1																																X
UC2																									X							
UC3																						X	X	X			X					
UC4		X	X	X	X									X																		
UC5		X	X	X	X									X																		
UC6		X	X	X	X																											
UC7																																
UC8						X	X																									
UC9						X	X																									
UC10		X	X	X	X									X																		
UC11																						X	X	X		X						
UC12																						X	X	X		X						
UC13																						X	X	X		X						
UC14																		X	X	X	X											
UC15																		X	X	X	X											
UC16																		X	X	X												
UC17																		X	X	X	X									X		
UC18													X	X	X	X	X															
UC19													X	X	X	X	X															
UC20																														X		
UC21		X	X	X																												
UC22								X		X	X	X																				
UC23								X		X	X																					
UC24										X	X	X																				
UC25																														X		
UC26																														X		
UC27																															X	
UC28																									X						X	
UC29																									X						X	
UC30																									X							
UC31									X																			X				

B. SYSTEM OPERATION CONTRACTS

Name:	**CheckTableStatus**
Responsibilities:	Determines which tables are clean, dirty, occupied, or reserved
Cross References:	Use Cases: CheckTableStatus
Exceptions:	None
Preconditions:	There are tables.
Postconditions:	The current conditions of the tables were updated and displayed.
Name:	PlaceOrder
Responsibilities:	Records customers' orders.
Cross References:	Use Cases: PlaceOrder
Exceptions:	None
Preconditions:	There are menu items to order.
Postconditions:	Orders were recorded by the system.

Name:	**AuthenticateUser**
Responsibilities:	Checks the user logging into the system is an employee.
Cross References:	Use Cases: AuthenticateUser
Exceptions:	None
Preconditions:	There are users in the system database.
Postconditions:	**User was identified as either an employee or not.**

Name:	StockStats
Responsibilities:	Returns the quantities of items in the inventory.
Cross	Use Cases: StockStatcs

References:	
Exceptions:	None
Preconditions:	There is a list of inventory items.
Postconditions:	The amounts of each inventory item was displayed.

Name:	**ViewRestaurantStats**
Responsibilities:	Displays the popularity of menu items along with menu and restaurant statistics.
Cross References:	Use Cases: ViewRestaurantStats
Exceptions:	None
Preconditions:	There are menu items to be ordered.
Postconditions:	The frequency of each menu item ordered was displayed.

Name:	**EditMenuItems**
Responsibilities:	Adds and/or deletes menu options.
Cross References:	Use Cases: EditMenuItems
Exceptions:	None
Preconditions:	There are menu items already in the system.
Postconditions:	The items to be added to the menu and deleted from the menu were added and deleted.

Name:	**ClockIn**
Responsibilities:	Records the time an employee begins working.
Cross References:	Use Cases: ClockIn
Exceptions:	None

Preconditions:	An employee is authenticated and logged in.
Postconditions:	The time the employee clocked in was recorded in the system.

Name:	**ClockOut**
Responsibilities:	Records the time an employee ends a work shift.
Cross References:	Use Cases: ClockOut
Exceptions:	None
Preconditions:	An employee is clocked in.
Postconditions:	The employee's working hours were adjusted to with the recorded clock out time, with wages adjusted.

Name:	**AddEmployee**
Responsibilities:	Creates a new employee profile in the system.
Cross References:	Use Cases: AddEmployee
Exceptions:	None
Preconditions:	The employee is not already in the system and the manager is logged in.
Postconditions:	A new profile was added to the employee roster.

Name:	**RemoveEmployee**
Responsibilities:	Deletes an existing employee profile from the system.
Cross References:	Use Cases: RemoveEmployee
Exceptions:	None
Preconditions:	The employee's profile to be deleted is already in the system.
Postconditions:	The employee's profile was removed from the list of employees in the system

Name:	**UpdateEmployee**
Responsibilities:	Updates the information for the employee profiles in the system.
Cross References:	Use Cases: UpdateEmployee
Exceptions:	None
Preconditions:	There are employee profiles in the system.
Postconditions:	Old employee information was replaced by newer information.

Name:	**ReleasePayment**
Responsibilities:	Pays employees the amount earned and updates their payroll
Cross References:	Use Cases: ReleasPayment
Exceptions:	None
Preconditions:	There are employees with wages and payroll information.
Postconditions:	The payroll was reset after a payment to the respective employees and statistics recorded.

B. MATHEMATICAL MODEL

The application requires some very basic algorithms in order to calculate various restaurant statistics. Those are as follows.

Payroll

The system will calculate payroll statistics. The system will access the hours the employee worked, multiply the number of hours by their hourly wage, and then record how much the employee was paid and reset the employee’s hours to zero.

Check Statistics

The manager will be allowed to check the menu trend and most popular hours that that customers visit the restaurant. The system will return the most popular food items by returning their sales per week. Also, the system will return the average customers per hour on any given day.

Check Stock

The system will simply access the current amount of stock and return the values. It will also give recommended amounts to order based on the trends in menu items and the ingredients they are composed of.

V. BUSINESS POLICIES

Below, we will list a various number of situations as well as conditions to these situations. These particular situations are relevant because they describe the conditions for several of our use cases.

1. An employee will be deleted from the system in various situations. One situation would be that the employee was fired due to not following the company policies such as being continuously being late or not showing up to work without having the shift covered. Another situation would be if an employee has not worked for more than two months.

2. An employee may remove an item from a person's bill in the following cases:
   a. An employee ordered the incorrect item.
   b. The item was delivered to the customer in an inappropriate manner such as not being properly cooked or unsanitary.
   c. The item was delivered after an appropriate amount of wait time, and the customer complained. With the interview at Applebee's, it was learned that an appropriate amount of time varies on whether or not it is busy. The waiter uses his discretion as to whether or not the time the guests had waited for food was appropriate.

3. A guest can make a reservation as long as a phone number is left. A condition for this is to assure that a reservation is possible to be made. The amount of reservations that could be made is dependent on the time. For the AM shift, there can be a high limit on the reservations such as 5 per hour due to the low frequency of customers. For the PM shift, we will allow 2 reservations per hour. Once these reservations are made, tables that could accommodate the guests will be reserved a half hour prior, similar to Applebee's. If the guests do not arrive 15 minutes after the time the reservation was for, then the table will be given up to another party if it is during a PM shift. The guests that have made the reservation will be made aware of this policy. Also, it often occurs that a table cannot be reserved half hour prior due to having a 'full house' or having all the tables being occupied. Thus, there will potentially be a wait, and the guests will be made aware of this fact.

4. A guest can cancel a reservation. There are no restrictions as to whether or not you could cancel a reservation, but it is preferred if a reservation is canceled a half hour prior.

5. When a guest enters the restaurant, ideally the guest shall be sat in a short amount of time. Again, this is dependent on the particular time of the day and week. The business policy will be that if all tables are occupied, then the wait will begin at approximately 10 minutes for parties of a size less than or equal to 6. Any guest after the first party will have a weight time 5 minutes longer. Larger parties that consist of 7 or more guests will naturally have a longer wait due to the greater number of guests that need to be accommodated.

VII. INTERACTION DIAGRAMS

The formatting of the figures will vary between interaction diagrams due to the use of different software to compile them. Since some group members have Mac's and other members have PC's, the software used by the PC's did not contain a cross platform support for Mac's.

CheckTablesStatus

This case's main role is to check the status of the tables' in the restaurant. The first role is assigned to the Host and follows the Expert Doer Principle since the host is the first to learn the information needed for this particular use case. The Database sends back information to the DatabaseController. Then, the GUIController displays the TablesStatus. There exists high cohesion in that most objects do not have many responsibilities. When the Database is asked for the TableStatusRequest, there is a loop that checks all tables in the restaurant for the data (the data being the status of each table). This particular sequence of call is the most responsibility an object in this use case has.

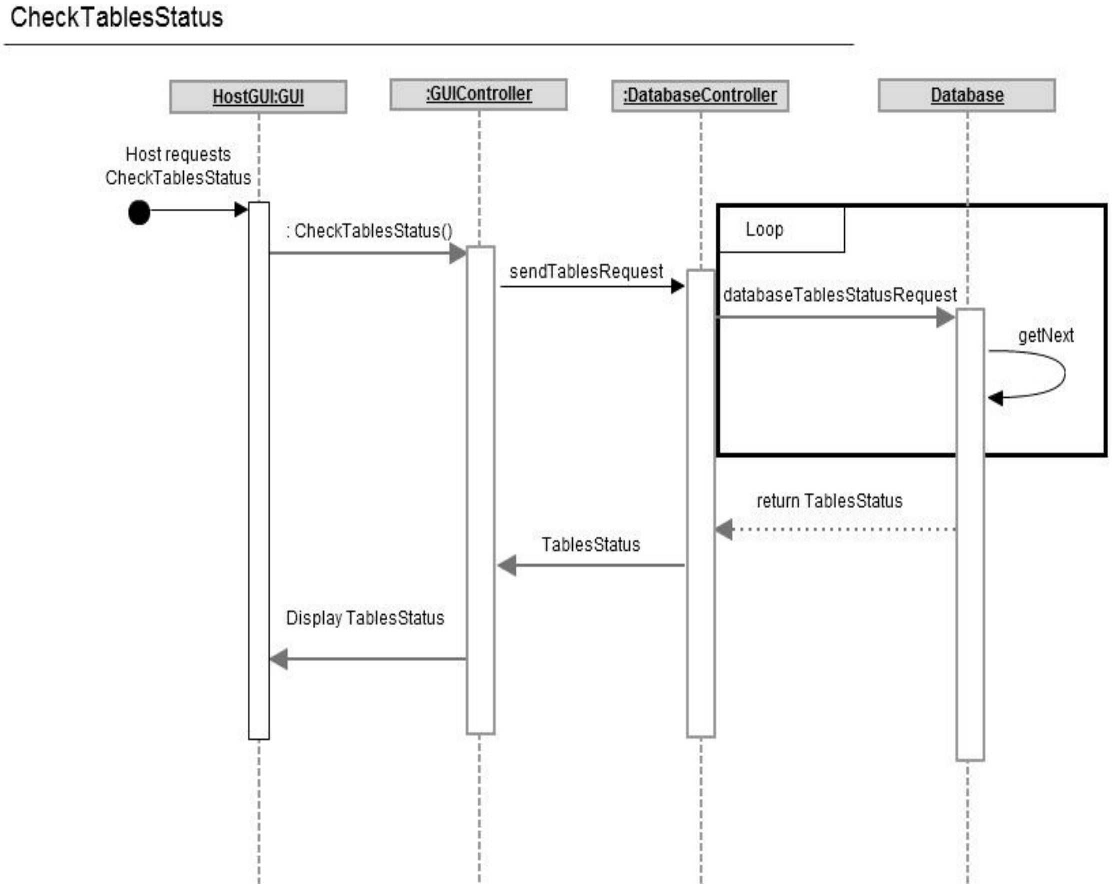


Figure 5.1: CheckTableStatus interaction diagram.

PlaceOrder

This interaction exhibits high cohesion as no object has many responsibilities. The Controller and has other functionalities when dealing in other use-cases and as a result it is not tasked with the responsibility of the specialized object OrderArchiver or Order. The coupling in this interaction is not low with respect to controller. It has the responsibility of communicating with each other object. DatabaseConnection exhibits expert doer principle because it retrieves data from the database; which is a system known only to this object.

The interaction for placing an order proceeds as followed:

User enter and order on the UserInterface which is then requested to be added through AddOrderRequest, which creates an Order. Then the order is sent to the controller, which verifies with that database that the order can be placed. At this point there are two scenarios. The first scenario is that the order is valid and then a loop is run until the order can be saved by the OrderArchiver. Finally, the Controller tells the UserInterface to prompt that the order was placed. The second scenario is when the order is not valid. When this occurs the Controller tells the UserInterface to prompt that the order was not placed.

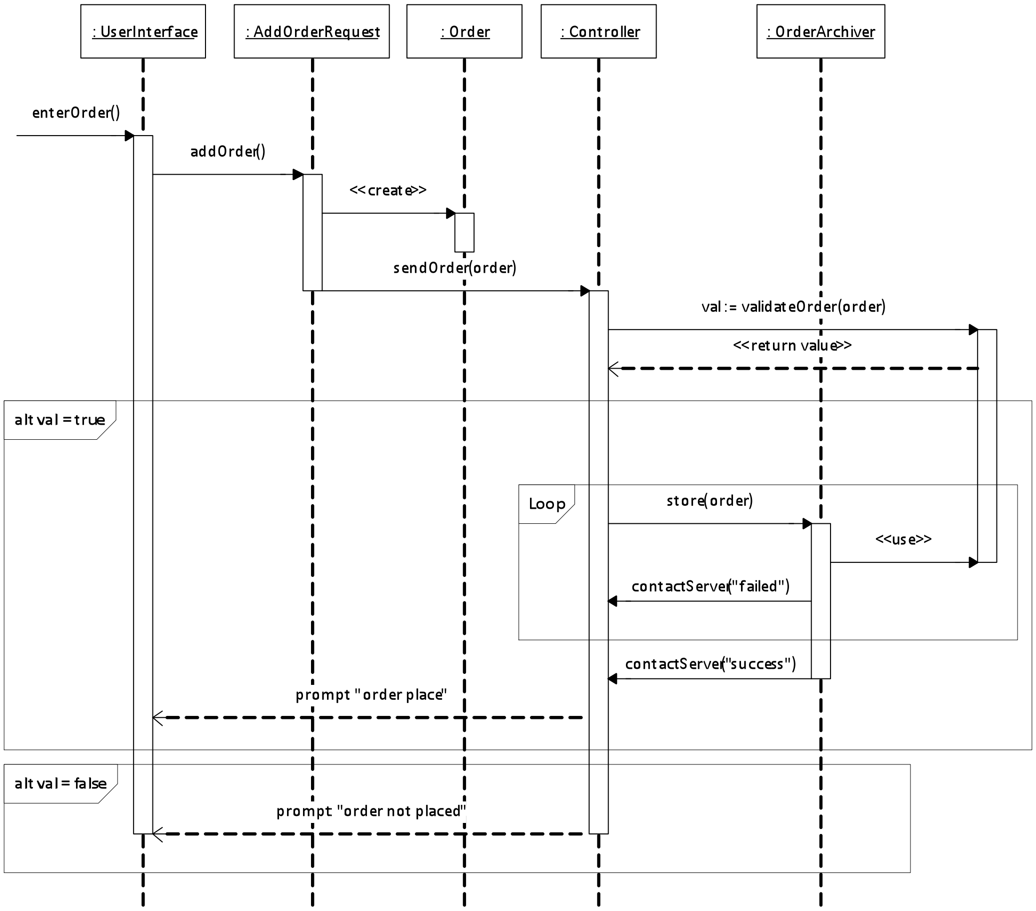


Figure 5.2: PlaceOrder interaction diagram.

UpdateEmployee

A call is made from the payroll interface when the UpdateEmployee button is pushed. The payroll controller would generate a form and receive an argument for the employee to be updated. This information would be utilized to populate a database query. Depending upon the search, the database will either return the employee object or notify the payroll controller of the failed attempt. Included in the employee object will be current information for the manager looking to update the information. This could include data such as name, address, phone number, wage level, etc.

When the database successfully returns an employee object and a modifiable form is generated by the Payroll controller, the user will then input the updated employee information. A validation object will check this information to ensure that it is correct before allowing modification of the actual employee object.

If validation is successful, the modified employee is passed back to the database. The database connection will return either successfully or unsuccessfully. An unsuccessful return , as is the case for unsuccessful employee data entry, results in a return back to the employee form.

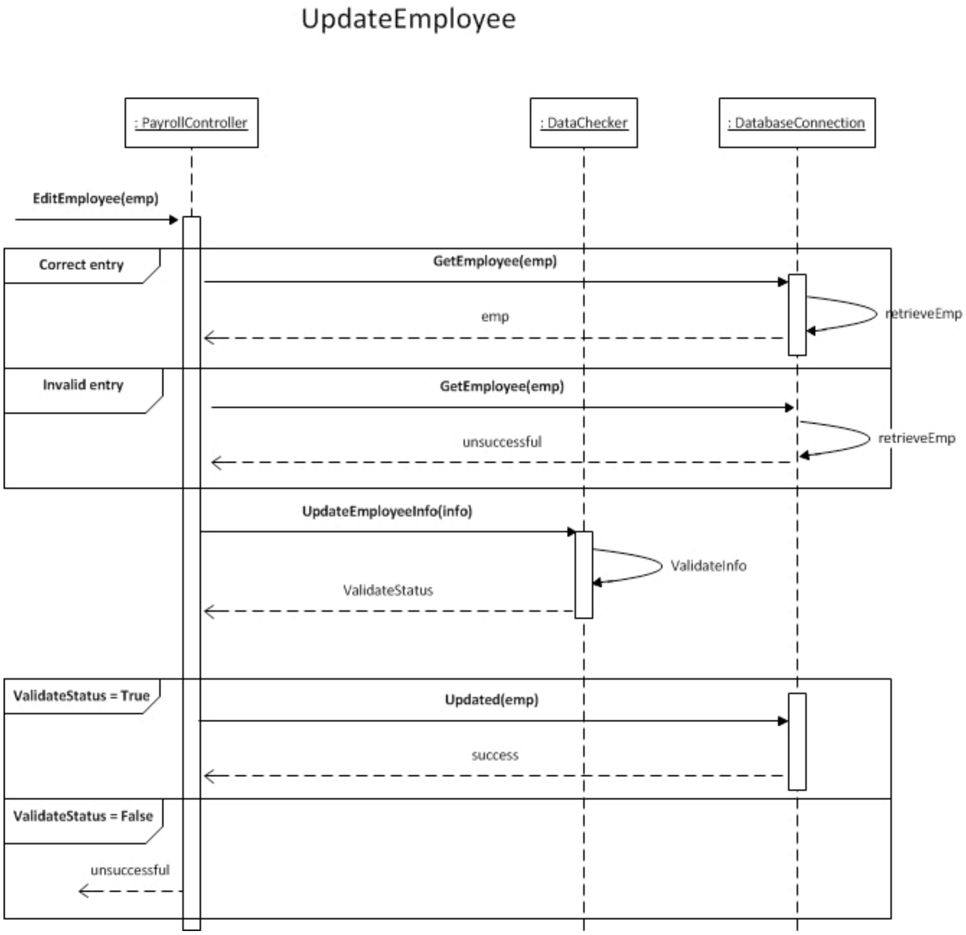


Figure 5.3: UpdateEmployee interaction diagram.

EditMenu

The interactions involved in editing the menu will be rather bulky if not split into subsequent interactions. With the principle of high cohesion, there is more focus on computational responsibilities. The function call to add to the menu only has concerns with adding to the current menu. Likewise, the delete function call only centers on the deletion of a menu item. As for the objects themselves, a more expert doer principle applies with the "MenuController" object doing most of the work in manipulating the information for changing the menu. However, this allows for a loose coupling between the menu and the food items. This ensures a change to food items does not drastically affect the menu itself.

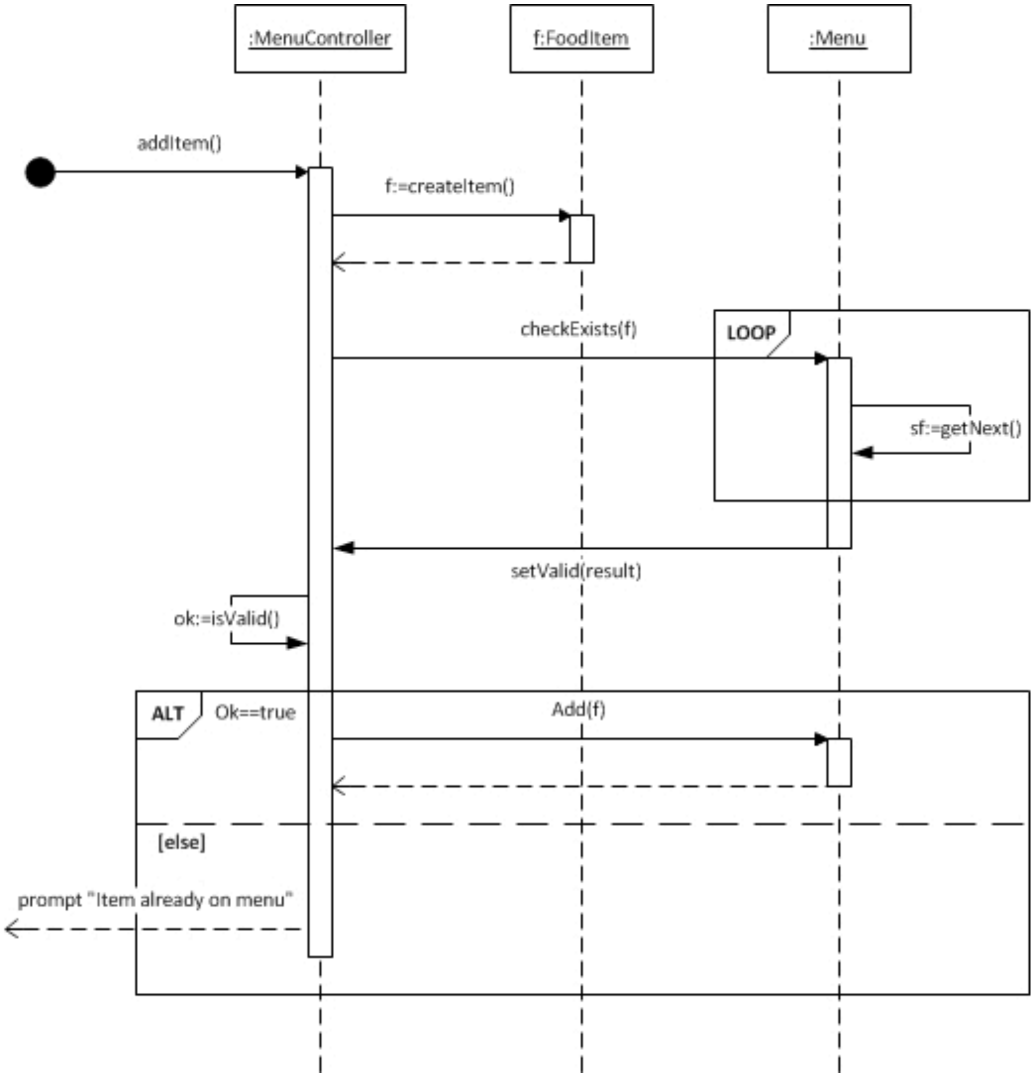


Figure 5.4a: AddItem interaction diagram.

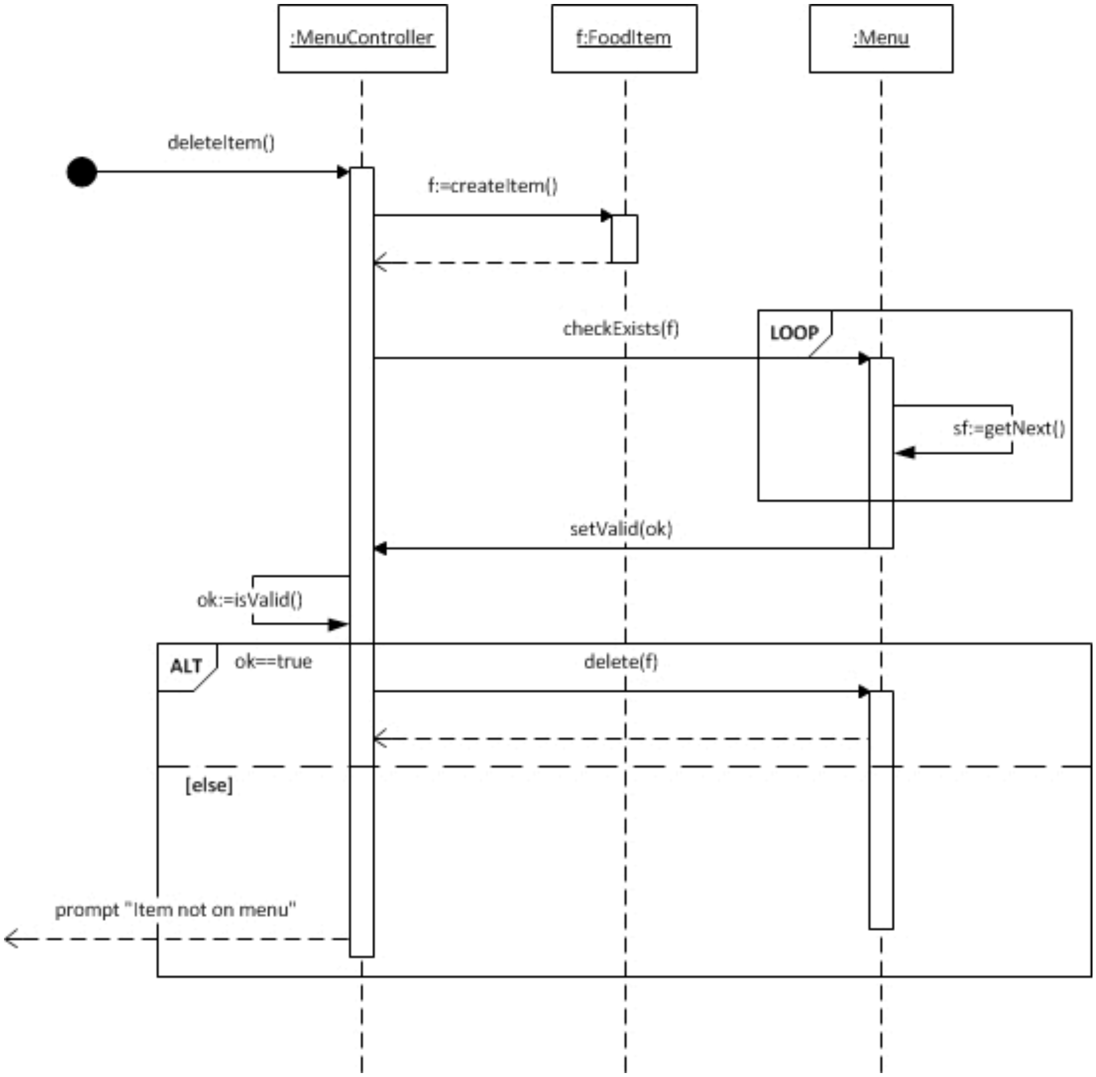


Figure 5.4b: DeleteItem interaction diagram.

CheckStats

This interaction diagram shows how the system distributes responsibility to various software components. The Main Controller is used in various other interactions. The Stat Controller keeps track of stats by reading and writing them to the database as well as analyzing them for graphical display.

The interaction for checking the restaurant stats goes as follows: The user (manager) requests to see the stats with GetRestaurantStats(s), s being the specific stat to be displayed (menu item trend, customer trends, etc.). This is passed to the Main Controller with getStats(s), and further goes to the Stat Controller through sendStatRequest. The Stat Controller performs a database request to get the pertinent data. It is then returned to the Stat Controller for analysis specific to the type of data requested. This then returns to the main controller where it gets displayed to the user.

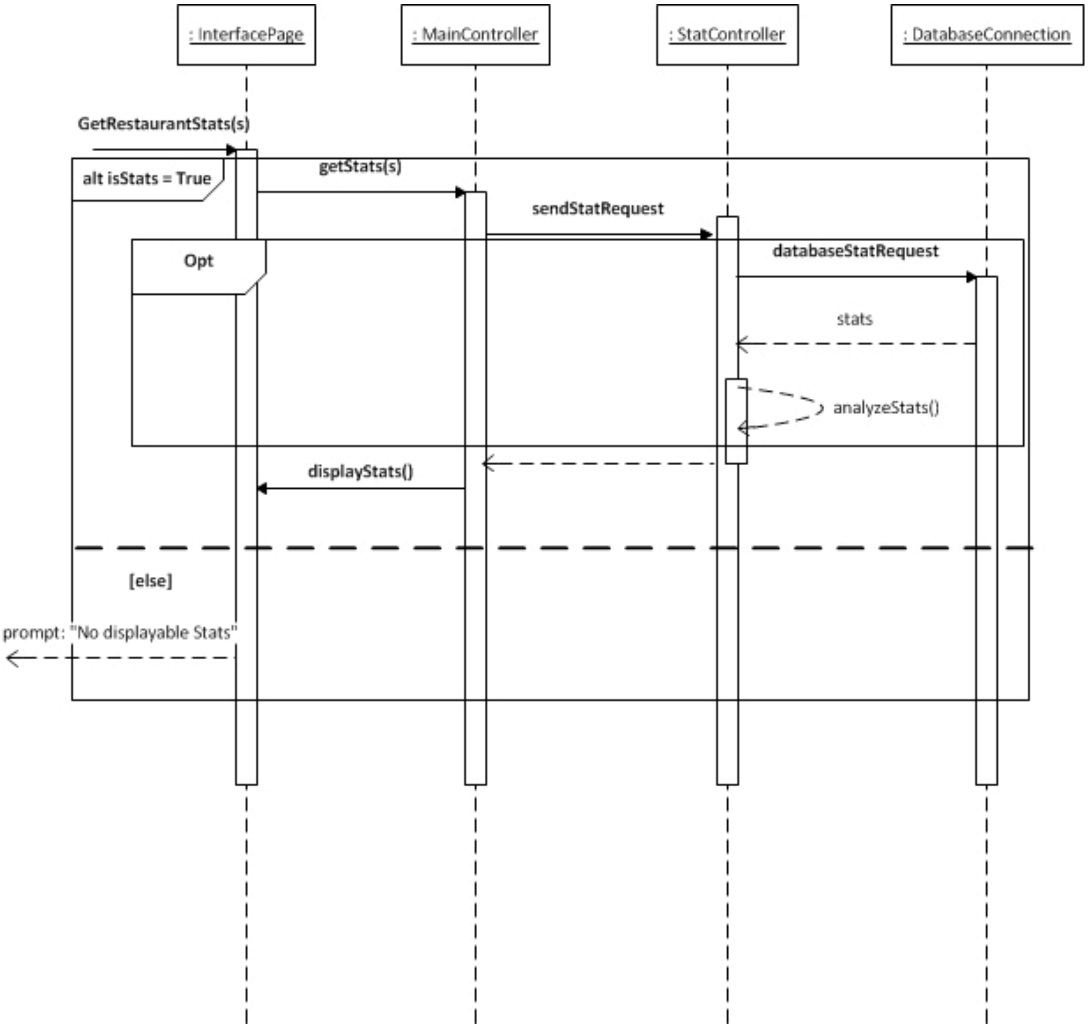
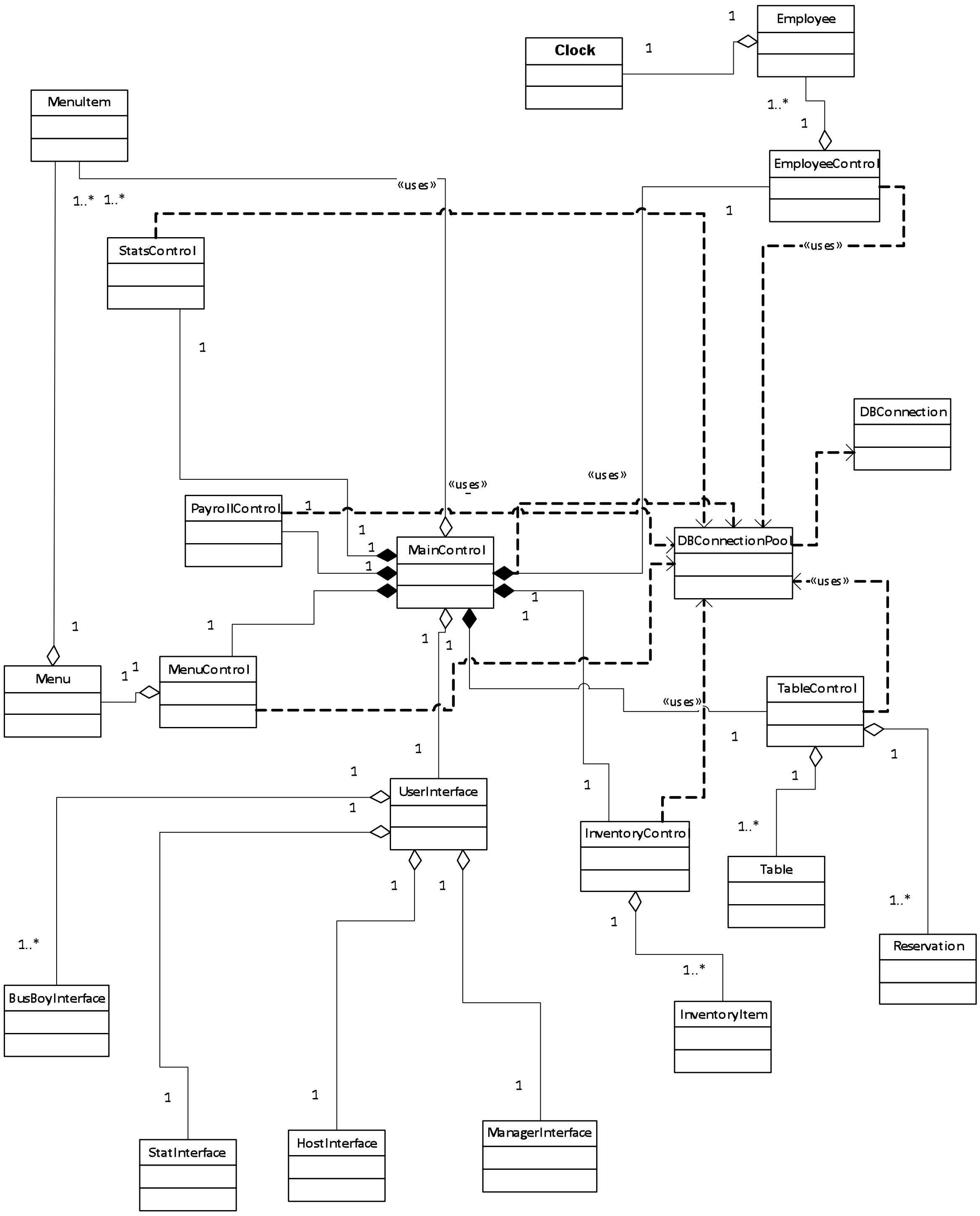
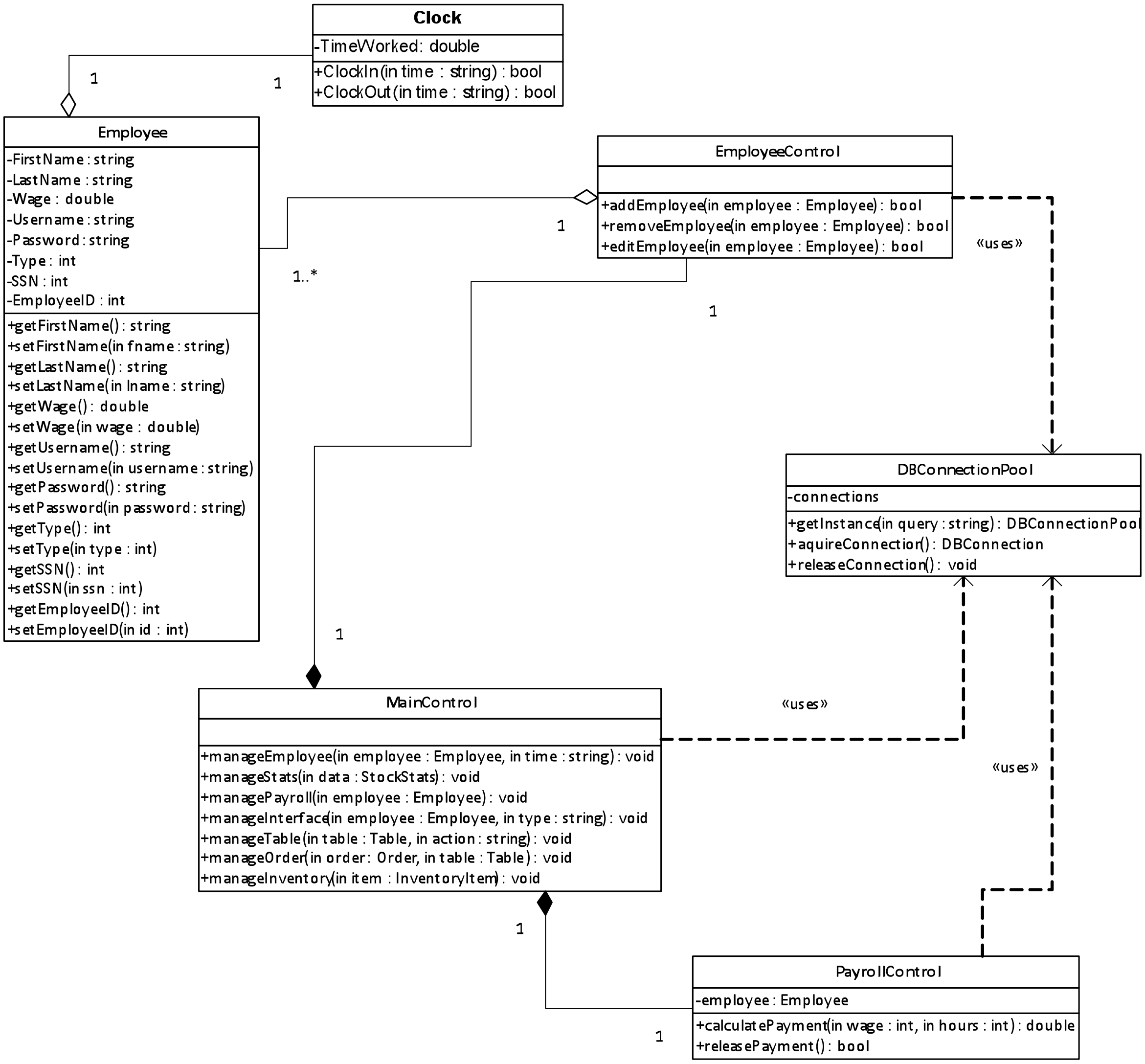


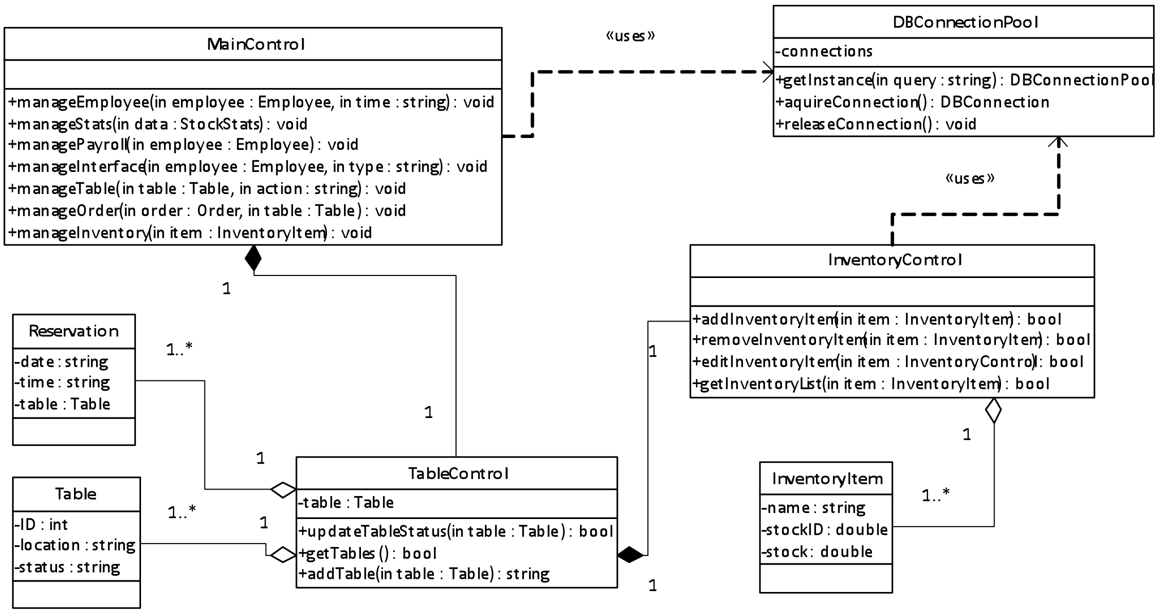
Figure 5.5: CheckStats interaction diagram.

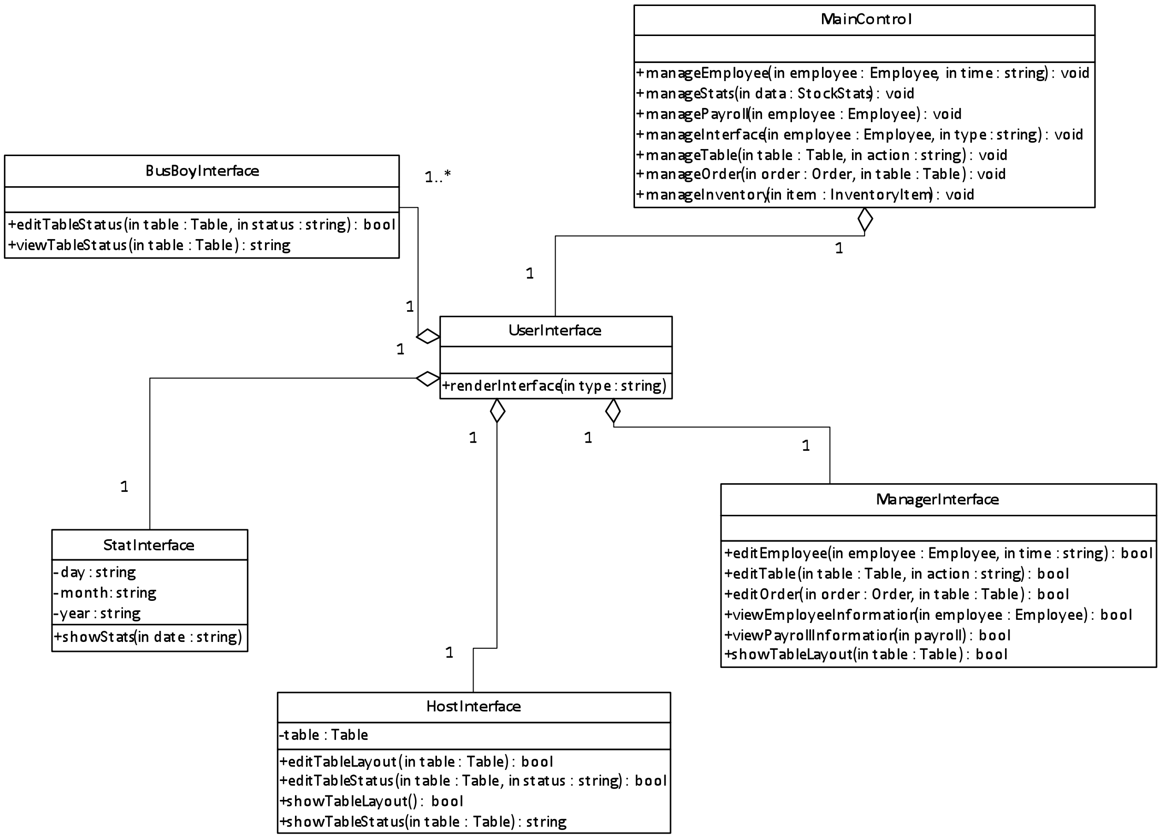
VIII. CLASS DIAGRAM AND INTERFACE SPECIFICATION

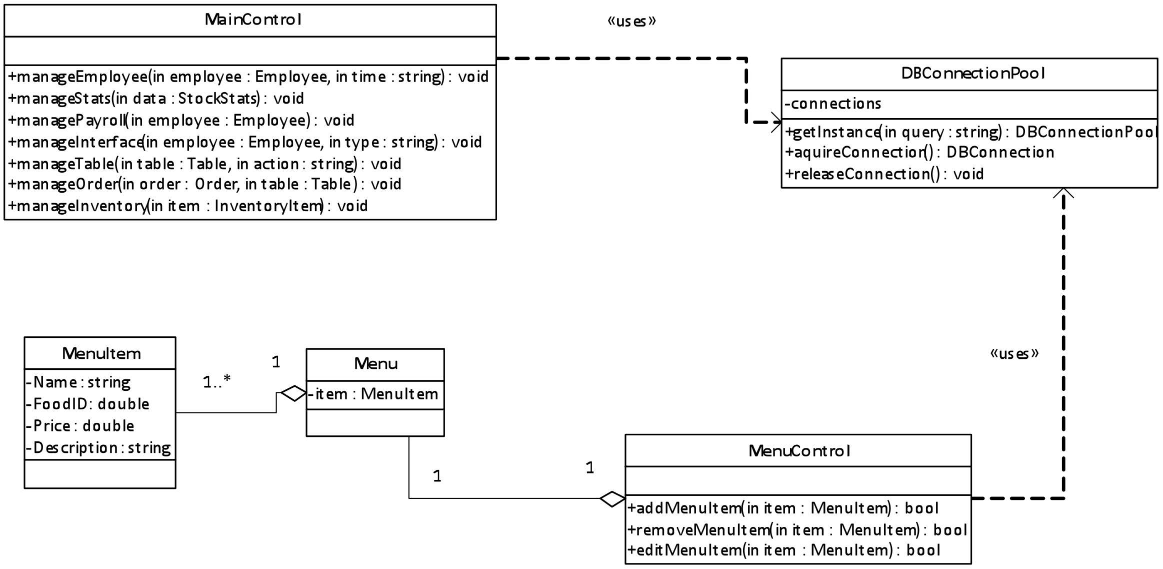
A. CLASS DIAGRAM

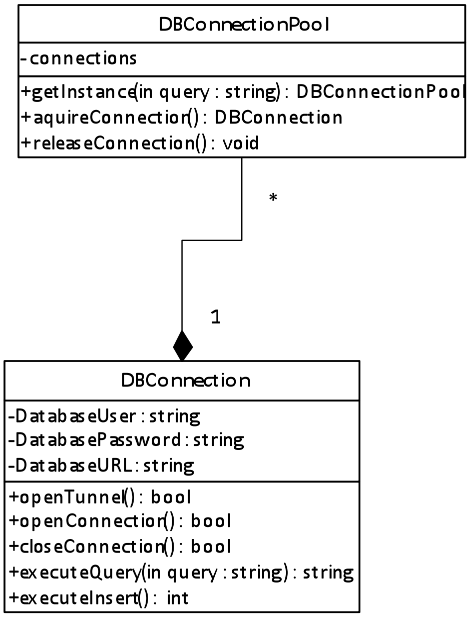



Clock
-TimeWorked: double
+ClockIn(in time : string) : bool
+ClockOut(in time : string) : bool
1
1
Employee
-FirstName : string
-LastName : string
-Wage : double
-Username : string
-Password : string
-Type : int
-SSN : int
-EmployeeID : int
+getFirstName() : string
+setFirstName(in fname : string)
+getLastName() : string
+setLastName(in lname : string)
+getWage() : double
+setWage(in wage : double)
+getUsername() : string
+setUsername(in username : string)
+getPassword() : string
+setPassword(in password : string)
+getType() : int
+setType(in type : int)
+getSSN() : int
+setSSN(in ssn : int)
+getEmployeeID() : int
+setEmployeeID(in id : int)
EmployeeControl
+addEmployee(in employee : Employee) : bool
+removeEmployee(in employee : Employee) : bool
+editEmployee(in employee : Employee) : bool
1
1..*
1
«uses»
DBConnectionPool
-connections
+getInstance(in query : string) : DBConnectionPool
+aquireConnection() : DBConnection
+releaseConnection() : void
1
MainControl
+manageEmployee(in employee : Employee, in time : string) : void
+manageStats(in data : StockStats) : void
+managePayroll(in employee : Employee) : void
+manageInterface(in employee : Employee, in type : string) : void
+manageTable(in table : Table, in action : string) : void
+manageOrder(in order : Order, in table : Table) : void
+manageInventory(in item : InventoryItem) : void
«uses»
«uses»
1
PayrollControl
-employee : Employee
+calculatePayment(in wage : int, in hours : int) : double
+releasePayment() : bool
1


MainControl
+manageEmployee(in employee : Employee, in time : string) : void
+manageStats(in data : StockStats) : void
+managePayroll(in employee : Employee) : void
+manageInterface(in employee : Employee, in type : string) : void
+manageTable(in table : Table, in action : string) : void
+manageOrder(in order : Order, in table : Table) : void
+manageInventory(in item : InventoryItem) : void
«uses»
DBConnectionPool
-connections
+getInstance(in query : string) : DBConnectionPool
+aquireConnection() : DBConnection
+releaseConnection() : void
«uses»
InventoryControl
+addInventoryItem(in item : InventoryItem) : bool
+removeInventoryItem(in item : InventoryItem) : bool
+editInventoryItem(in item : InventoryControl) : bool
+getInventoryList(in item : InventoryItem) : bool
Reservation
-date : string
-time : string
-table : Table
Table
-ID : int
-location : string
-status : string
TableControl
-table : Table
+updateTableStatus(in table : Table) : bool
+getTables() : bool
+addTable(in table : Table) : string
InventoryItem
-name : string
-stockID : double
-stock : double
1
1..*
1
1
1..*
1
1
1
1
1..*


MainControl
+manageEmployee(in employee : Employee, in time : string) : void
+manageStats(in data : StockStats) : void
+managePayroll(in employee : Employee) : void
+manageInterface(in employee : Employee, in type : string) : void
+manageTable(in table : Table, in action : string) : void
+manageOrder(in order : Order, in table : Table) : void
+manageInventory(in item : InventoryItem) : void
BusBoyInterface
+editTableStatus(in table : Table, in status : string) : bool
+viewTableStatus(in table : Table) : string
1..*
1
1
1
1
UserInterface
+renderInterface(in type : string)
1
1
1
1
1
StatInterface
-day : string
-month : string
-year : string
+showStats(in date : string)
ManagerInterface
+editEmployee(in employee : Employee, in time : string) : bool
+editTable(in table : Table, in action : string) : bool
+editOrder(in order : Order, in table : Table) : bool
+viewEmployeeInformation(in employee : Employee) : bool
+viewPayrollInformation(in payroll) : bool
+showTableLayout(in table : Table) : bool
1
HostInterface
-table : Table
+editTableLayout(in table : Table) : bool
+editTableStatus(in table : Table, in status : string) : bool
+showTableLayout() : bool
+showTableStatus(in table : Table) : string


MainControl
+manageEmployee(in employee : Employee, in time : string) : void
+manageStats(in data : StockStats) : void
+managePayroll(in employee : Employee) : void
+manageInterface(in employee : Employee, in type : string) : void
+manageTable(in table : Table, in action : string) : void
+manageOrder(in order : Order, in table : Table) : void
+manageInventory(in item : InventoryItem) : void
«uses»
DBConnectionPool
-connections
+getInstance(in query : string) : DBConnectionPool
+aquireConnection() : DBConnection
+releaseConnection() : void
«uses»
MenuItem
-Name : string
-FoodID: double
-Price : double
-Description : string
1..*
1
Menu
-item : MenuItem
1
1
MenuControl
+addMenuItem(in item : MenuItem) : bool
+removeMenuItem(in item : MenuItem) : bool
+editMenuItem(in item : MenuItem) : bool


DBConnectionPool
-connections
+getInstance(in query : string) : DBConnectionPool
+aquireConnection() : DBConnection
+releaseConnection() : void
*
1
DBConnection
-DatabaseUser : string
-DatabasePassword : string
-DatabaseURL : string
+openTunnel() : bool
+openConnection() : bool
+closeConnection() : bool
+executeQuery(in query : string) : string
+executeInsert() : int

The class diagrams that are presented above show first the overall class diagram with the class names and the interaction between the classes. As seen above, the main hub of communication is the MainControl class as well as the DBConnectionPool, which links to the DBConnection, from which all of the data is stored and retrieved. The UserInterface is also a main hub from which incoming requests from every other interface is transmitted to the MainControl and then delegated to the specified controllers (ie. OrderControl, InventoryControl, EmployeeControl).

The subsequent class diagrams are split logically by specialization of operations where each controller is specified within its own class diagram. There are also diagrams that were incorporated to show logical connections between all of the classes. For example, there is a class diagram that refers to all of the interfaces which links them to the MainControl, which is at the center of the program.

The DBConnectionPool and the DBConnection were show in a separate class diagram because they represent the design pattern of ObjectPool and DBConnection does not need to be re-iterated in every class diagram.

B. DATA TYPES AND OPERATION SIGNATURES

MainController

Attributes:

Operations:

+manageEmployee(in employee:Employee, in time:string) : void
//Control over employee profiles and their working hours
+manageStats(in data:StockStats) : void
//Control over the data for the restaurant's food orders and customer actions
+managePayroll(in employee:Employee) : void
//Control of the financial aspects of each employee including payout and wages
+manageInterface(in employee:Employee, in type:string) : void
//Control of the user interfaces of all restaurant personnel.
+manageTable(in table:Table, in action:string) : void
//Control over all changes to table statuses, number, and location.
+manageOrder(in order:Order, in table:Table) : void
//Control over all orders including order placement and cancellation
+manageInventory(in item:InventoryItem) : void
//Control over all inventory items and quantities

DBConnection

Attributes:

-DatabaseUser:string
//The username for all employee logins
-DatabasePassword:string
//The password for all employee logins

```
        -DatabaseURL:string
            //The location of the database

    Operations:
        +openConnection() : bool
            //Connects to the database
        +closeConnection() : bool
            //Disconnections from the database
        +queryDatabase( in query:string ) : string
            //Sends a query to the database to retrieve information

Archiver
    Attributes:
        -DBConnection:DBConnection
            //The connection with database

    Operations:
        +storeData( in query:string ) : bool
            //Stores data from query to the database

PayrollControl
    Attributes:
        -employee:Employee
            //The employee referred to for payroll

    Operations:
        +calculatePayment( in wage:int, in hours:int ) : double
            //Determines the amount earned for the employee
        +releasePayment() : bool
            //Pays the employee the amount earned and resets hours worked

Clock
    Attributes:
        -TimeWorked : double
            // amount of time an employee has worked

    Operations:
        +ClockIn( in time:string ) : bool
            // records the time when an employee starts working
        +ClockOut( in time:stirng ) : bool
            //records the ending time of an employee's work shift

EmployeeControl
    Attribute:
```

- Operations:
    - +addEmployee(in employee:Employee) : bool
        - //Puts a new employee profile into the database
    - +removeEmployee(in employee:Employee) : bool
        - //Deletes and existing employee profile from the database
    - +editEmployee(in employee:Employee) : bool
        - //Changes the information in the employee profile in the database

Employee

- Attributes:
    - -FirstName:string
        - //Person’s first name
    - -LastName:string
        - //Person’s last name
    - -Wage:double
        - //The amount earned per hour worked
    - -Username:string
        - //Username for logging in
    - -Password:string
        - //Password for logging in
    - -Type:int
        - //Occupational role in the restaurant
    - -SSN:int
        - //Social Security Number
    - -EmployeeID:int
        - //An identification number

- Operations:
    - +getFirstName() : string
        - //Returns the employee’s first name
    - +setFirstName(in fname:string)
        - //Changes/makes first name to fname
    - +getLastName() : string
        - //Returns the employee’s last name
    - +setLastName(in lname:string)
        - //Changes/makes last name lname
    - +getWage() : double
        - //returns the employee’s wage
    - +setWage(in wage:double)
        - //Changes/makes the wage as specified
    - +getUsername() : string
        - //Returns the employee’s username
    - +setUsername(in username:string)

```
            //Changes/makes the username as specified
        +getPassword() : string
            //Returns the employee's password
        +setPassword( in password:string)
            //Changes/makes the password as specified
        +getType() : int
            //Returns the employee's occupational role
        +setType( in type:int )
            //Changes/makes the employee's occupational role as specified
        +getSSN() : int
            //Returns the social security number of an employee
        +setSSN( in ssn:int )
            //Changes/makes the social security number a 9 digit integer
        +getEmployeeID() : int
            //Returns the employee's identification number
        +setEmployeeID( in id:int )
            //Changes/makes the employee's identification number as specified.

MenuItem
    Attributes:
        -Name : string
            // Name of food on the menu
        -FoodID : double
            // Identification number for food
        -Price : double
            // Ordering price of the food
        -Description : string
            // Describes the food item

    Operations:

Order
    Attributes:
        -FoodItem : FoodItem
            //A food item from the menu
        -Progress : FoodProgress
            //The state of progress of the ordered food
    Operations:

OrderControl
    Attributes:
        -numOfOrders : double
            //Keeps track of how many orders there are
```

- Operations:
  - +displayOrder()
    - //Shows the orders to the chef and customer
  - +invalidOrder(in order:Order) : bool
    - //Tells whether the order can be made depending on inventory stocks
  - +addOrder(in order:Order)
    - //Adds an order from the menu to be cooked
  - +editOrder(in order:Order)
    - //Edits the orders made to the customers needs (ex. Adding cheese)
  - +cancelOrder(in order:Order)
    - //Cancels an order if made within 2 minutes of cancellation

FoodProgress

- Attributes:
  - -OrderStatus : string
    - //variable that stores whether an order is started, being prepared, or finished
- Operations:
  - +getOrderStatus()
    - //Returns the whether an order is started, being prepared, or finished
  - +setOrderStatus(in status:string)
    - //Makes an order with the status specified

StatsControl

- Attributes:
  - -day : string
    - //The day
  - -week : string
    - //The week
  - -year : string
    - //The year
- Operations:
  - +showDailyStats(in date:string) : void
    - //displays the financial statistics for every day starting with the inputted date
  - +showWeeklyStats(in date:string) : void
    - //displays the financial statistics by weeks starting with the inputted date
  - +showMonthlyStats(in date:string) : void
    - //displays the financial statistics by months starting with the inputted date
  - +showYearlyStats(in date:string) : void
    - //displays the financial statistics by year starting with the inputted date

InventoryItem

- Attributes:

```
          -Name : string
               //name of food item in inventory
          -StockID : double
               //identification number for food item in inventory
          -Quantity : double
               //amount of the food item in the inventory

     Operations:


InventoryControl
     Attributes:

     Operations:
          +addInventoryItem( in item:string )
               //Adds an new item to the inventory
          +removeInventoryItem( in item:string )
               //Deletes an item from the inventory
          +addStock( in stock:int, in item:string )
               //Increases the quantity of an item in the inventory
          +removeStock( in stock:int, in item:string )
               //Reduces the quantity of an item in the inventory

Reservation
     Attributes:
          -date:string
               //The date of the reservation
          -time:string
               //The time of reservation
          -table:Table
               //The table to be reserved
     Operations:

Table
     Attributes:
          -ID:int
               //Table number
          -location:string
               //Coordinates of table in restaurant
          -status:string
               //Table status (dirty, reserved, occupied, clean)
     Operations:

TableControl
```

```
        Attributes:
            -table:Table
                //A table in the restaurant
        Operations:
            +editTableStatus( in table:Table ) : bool
                //Changes the status of a table
            +editTableLayout( in table:Table ) : bool
                //Moves a table to a different location
            +getTableStatus( in table:Table ) : string
                //Returns the status of a table
            +getTableLayout() : Table
                //Returns the layout of the tables
Menu
        Attributes:
            -item:MenuItem
                //An item on the menu
        Operations:

Menu Control
        Attributes:

        Operations:
            +addMenuItem( in item:MenuItem ) : bool
                //Adds an item to the menu
            +removeMenuItem( in item:MenuItem ) : bool
                //Removes an item from the menu

BusBoyInterface
        Attributes:

        Operations:
            +editTableStatus( in table:Table, in status:string ) : bool
                //Changes the table to clean
            +viewTableStatus( in table:Table ) : string
                //Returns the status of a table (dirty, reserved, occupied, clean)

CustomerInterface
        Attributes:

        Operations:
            +placeOrder( in order:Order )
                //Make an order from the menu and submit it to the chef.
            +makeReservation( in table:Table, in date:string )
                //Reserve a table(s) in the restaurant for a specified date.
```

```
          +payBill( in bill:double, in amount:double, in type:string )
               //Pays the final bill with either cash or credit card.
          +viewOrderStatus( in order:Order ) : bool
               //Shows the progress of an order
          +viewBill( in bill:string ) : bool
               //Shows the final bill
          +viewReservationList()
               //Shows all the reservations made

StatInterface
     Attributes:
          -day : string
               //The day
          -month : string
               //The month
          -year : string
               //The year

     Operations:
          +showStats( in date:string )
               //Displays the financial and customer statistics for the inputted date.

HostInterface
     Attributes:
          -table : Table
               //Represents an actual table in the restaurant.

     Operations:
          +editTableLayout( in table:Table ) : bool
               //Moves the selected table to a specified new location.
          +editTableStatus( in table:Table, in status:string ) : bool
               //Denotes the selected table as dirty, reserved, occupied, or clean.
          +showTableLayout() : bool
               //Displays the arrangement of tables in the restaurant
          +showTableStatus( in table:Table ) : string
               //Returns the status of a table

ManagerInterface
     Attributes:

     Operations:
          +editEmployee( in employee:Employee, in time:string ) : void
               //Add ore remove employees
          +82ditable( in table:Table, in action:string ) : void
```

//Change all aspects of tables including location and status

+editOrder(in order:Order, in table:Table) : void

//Change orders to add or remove items and cancel orders as needed.

+viewEmployeeInformation(in employee:Employee) : bool

//Shows the profile of an employee

+viewPayrollInformation(in employee:Employee) : bool

//Shows the payroll of an employee

+showTableLayout(in table:Table) : bool

//Displays the arrangement of tables in the restaurant

UserInterface

Attributes:

Operations:

+renderInterface(in type:string)

//Creates the screen for the user interfaces

C. TRACEABILITY MATRIX

Below is the traceability matrix showing all the classes that evolved from the domain concepts. Because of the immensity of classes and concepts, the matrix has been divided and placed on several pages.

The simplest changes from the concepts to the classes include the shortening of names, as seen from the domain concept *UpdateOrderStatusRequest* and the class *FoodProgress*. Also, an *Archiver* class is made to control all class actions to and from the database.

However, the major development to the software classes from the domain concepts is the idea of an overview controller that oversees operations in many operations. In addition, there is a main controller to handle all the other controller classes. This is the reason many classes take on the responsibilities of several domain concepts. The controller classes include: *MenuControl, PayrollControl, StatsControl, OrderControl, EmployeeControl, InventoryControl, TableControl, UserInterface,* and *MainControl*.

Furthermore, some domain concepts are mapped out to more than one class. This is due to the use of interface classes, which allow the respective users to do specific actions as needed, like viewing a table's status or viewing an employee's profile information. As a result, there is a user interface class for each actor, and an overall user interface class to render all the displays.

	Software Classes										
Domain Concepts	Employee	Clock	EmployeeControl	PayrollControl	FoodItem	StockItem	StockStats	InventoryControl	StatsControl	BusBoyInterface	CustomerInterface
Employee	x										
Clock		x									
ClockOutRequest		x									
ClockInRequest		x									
AddEmployeeRequest			x								
RemoveEmployeeRequest			x								
UpdateEmployeeRequest			x								
ReleasePayment				x							
MenuItem					x						
InventoryItem						x					
ViewInventoryRequest							x				
AddInventoryItemRequest							x				
RemoveInventoryItemRequest							x				
AddInventoryRequest								x			
RemoveInventoryRequest								x			
PayBillRequest									x		x
ViewBillRequest											x
ViewPayrollRequest											x
UserInterface										x	
Menu											
AddMenuItemRequest											

RemoveMenuItemRequest											
Controller											
DatabseConnection											
Archiver											

D. DESIGN PATTERNS

The system is fairly simple in its implementation and as a result many design patterns would only complicate the implementation with little to no benefit.

In the case of Publisher-Subscriber, there is no object for which a change in the system would need many other objects to be informed. The lack of necessity for many objects to be informed of system changes is due to the implementation of the user interface, Java Swing. This interface, part of the java.swing package, inherently does not require publisher-subscriber since all actions are known by the interface package. In every case of implementation, there is only one object that would subscribe to the actions of an object denoted as a publisher and as a result it is not necessary to use this design pattern. Expansion is also not a viable reason for using the Publisher-Subscriber pattern because of the flow of information.

The Proxy design pattern is used by the Java Database Connection (JDBC), which includes the design pattern within it. Java's implementation of JDBC interfaces the MySQL connection that we are using and proxies it within the local client in order to protect the information that is being sent to and from the remote server. As a result, through the use of the JDBC, the proxy design pattern is thus implicitly used in our implementation.

The Object Pool pattern is used to implement the database connection. The connection to the database is used often and constantly creates and destroys a connection object, which causes an unnecessary overhead. When the functionality is expanded, an even larger overhead is caused due to the sheer amount of connections that would need to be made. The object pool pattern is a set of initialized objects that are kept within a pool and are called upon to be re-used. In essence, the object pool is a form of the more generic Factory design pattern, where an object is tasked to create objects. In this special case, an object has the task of returning the set of initialized objects and holding a pool of these objects created for re-use, rather than constructing and destroying the objects. The object pool provides greater efficiency in the case of our implementation because we use a database connection often to retrieve and store data within the database. Accordingly, the object pool allows for the management of the number of connection objects made.

The Flyweight pattern would have been used for our table implementation; however, this was not possible due to the time constraint and the complexity of the design. The flyweight pattern also is a specific type of the Factory pattern that allows for many objects that share similar states and behavior to be easily supported. The flyweight pattern would be used because of all the tables have certain states (clean, occupied, and dirty) in which they can be and the large amount of tables present in the restaurant, which would use a lot of memory. The flyweight pattern solves the memory usage issue because there would be a "flyweight" class that holds the extrinsic state of the tables and would have methods that would change the extrinsic states that vary from object to object. This pattern also allows for efficiency in creating tables and reusability of instances of the classes as in the object pool pattern.

Due to time constrains and prior implementation of certain aspects of the project, many design patterns could not be used. However, there may be some design patterns that were inherently used in our program but not recognized.

E. OBJECT CONSTRAINT LANGUAGE (OCL) CONTRACTS

context EmployeeControl::addEmployee(in employee:Employee) : bool

pre: Employee.oclIsUndefined()

post: result = not Employee.oclIsUndefined()

context EmployeeControl::removeEmployee(in employee:Employee) : bool

pre: not Employee.oclIsUndefined()

post: result = Employee.oclIsUndefined()

context EmployeeControl::editEmployee(in employee:Employee) : bool

pre: not Employee.oclIsUndefined()

post: result = Employee.oclIsNew()

inv: Employee.SSN → size() == 9

inv: Employee.Password → size() >= 6

context Clock

inv: self.TimeWorked >= 0

context FoodProgress

inv: let status : string = {“Pending”, “Preparing”, “Cooking”, “Done”} in

self.OrderStatus = status

context InventoryControl::addInventory(in item:InventoryItem) : bool

pre: InventoryItem.oclIsUndefined()

post: result = not InventoryItem.oclIsUndefined()

inv: InventoryItem.stock → size() >= 0

context InventoryControl::removeInventoryItem(in item:InventoryItem) : bool

pre: not InventoryItem.oclIsUndefined()

post: result = InventoryItem.oclIsUndefined()

inv: InventoryItem.stock → size() >= 0

context InventoryControl::editInventoryItem(in item:InventoryItem) : bool

pre: not InventoryItem.oclIsUndefined()

post: result = InventoryItem.oclIsNew()

inv: InventoryItem.stock → size() >= 0

context InventoryItem

inv: InventoryItem.name → size() >=0

IX. SYSTEM ARCHITECTURE AND SYSTEM DESIGN

A. ARCHITECTURE STYLES

Depending on the details of a particular project, an architectural style must be chosen. One type of architectural style is the Repository Architectural Style where subsystems access and modify a central repository. All subsystems are independent and the only interaction that occurs is through the central repository. For our particular project, this architectural style is not the best. This architectural style is heavily dependent on the central repository. If something would lead towards the central repository losing its data, the data of the whole system would be lost. Also, changes that we would wish to create would be difficult to implement.

Another architecture style is the MVC (Model/View/Controller) Architectural Style. This particular subsystem separates all the subsystem into three categories defined as the model subsystems, view subsystems, and the controller subsystems. The model subsystems role is to store the data of the application. The view subsystems display data to the user. The controller subsystems manage the interactions between user and system. In our implementation, this is the best architectural style. By using this style, we are able to make changes, and it would be easily implemented.

Finally, there is the Three-Tier Architectural Style. The subsystems are separated into three parts: the Interface Layer, the Application Logic Layer, and the Storage Layer. The Interface layer is defined as the User Interface or the boundary that interacts with the user. The Application Logic Layer has the task of controlling objects. Lastly, the Storage Layer constitutes as the database. By dividing up the subsystems into parts, we are able to create changed that would not affect the other parts.

B. IDENTIFYING SUBSYSTEMS

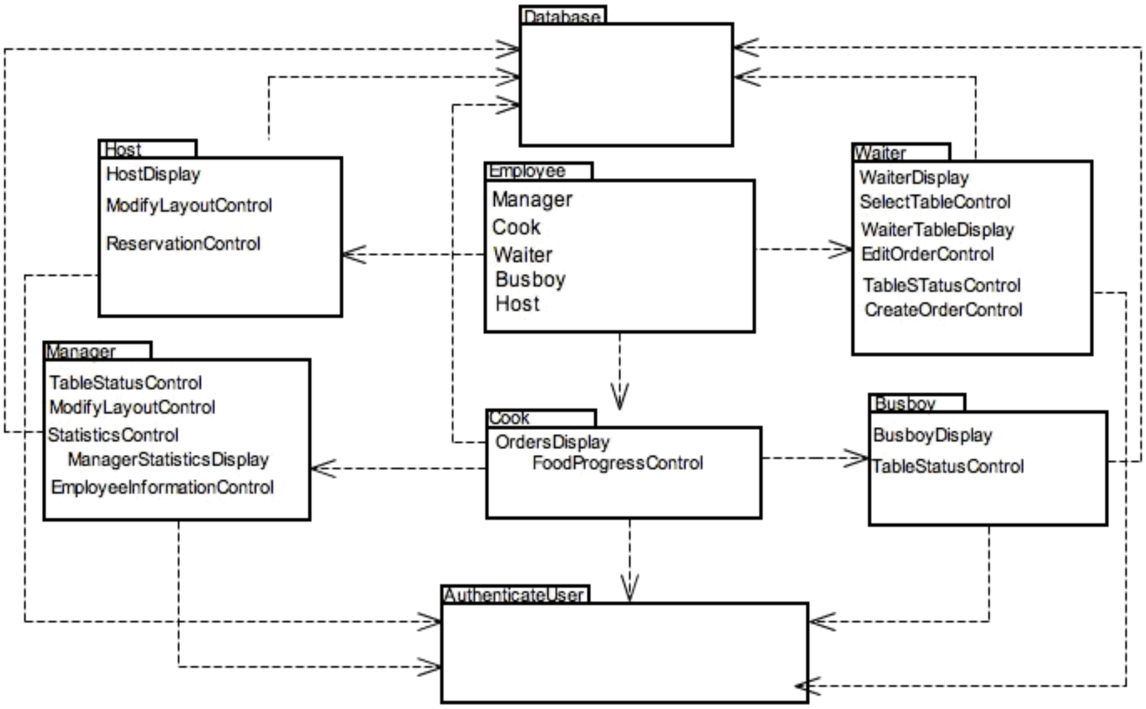


Figure 6: SubSystems

C. MAPPING SUBSYSTEMS TO HARDWARE

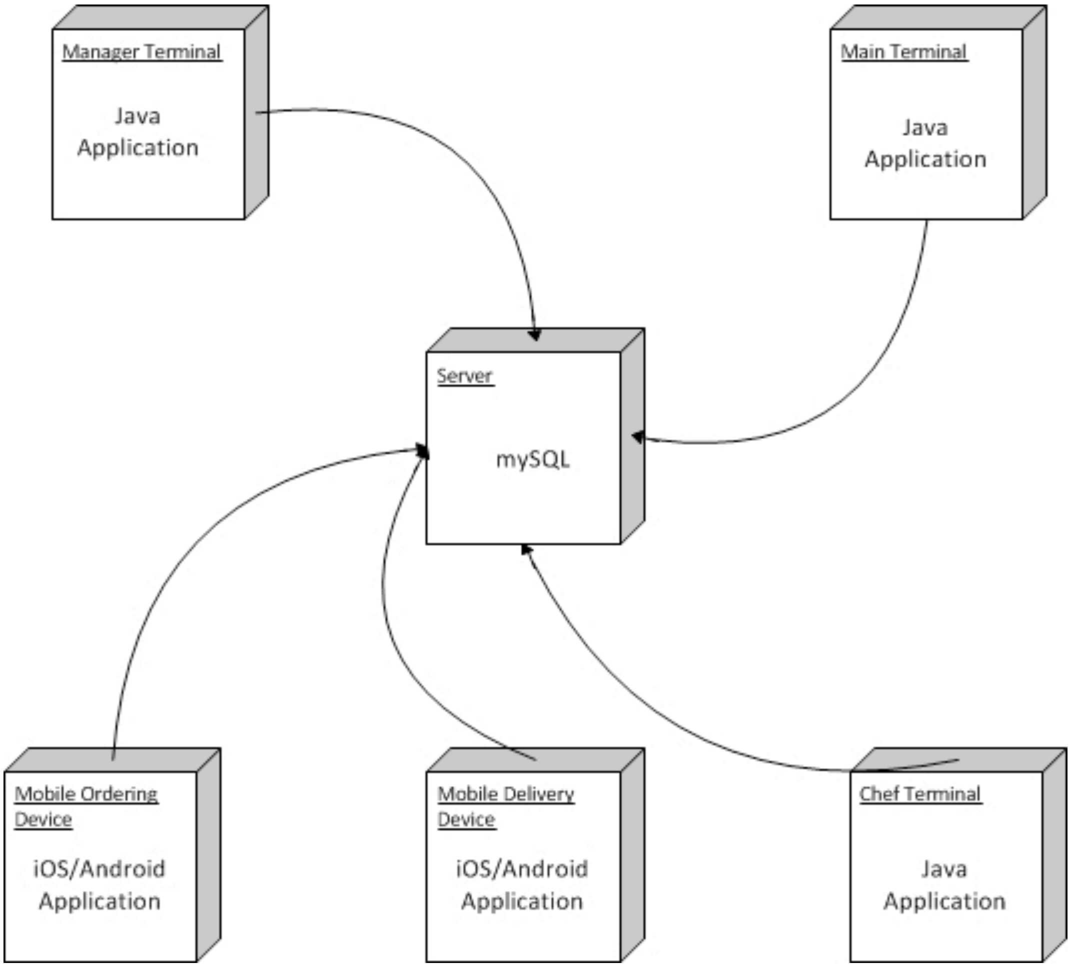


Figure 7: Subsystems to Hardware

D. PERSISTENT DATA STORAGE

The restaurant automation system will require storage and accessibility to the various parts of the system at different times. The data will be organized via a relational MySQL database to facilitate concurrent accessibility and ease of use. There is the potential that waiters, managers, hosts, chefs, delivery-boys and busboys will interact with their respective subsystems simultaneously, requiring reads and writes to the database concurrently. Thus, the database will have to have procedures in place to handle such traffic. Additionally, storing the data in a relational database will allow for efficient querying and manipulation of the data to the needs of each particular client application. The persistent data objects are shown in the schema below:

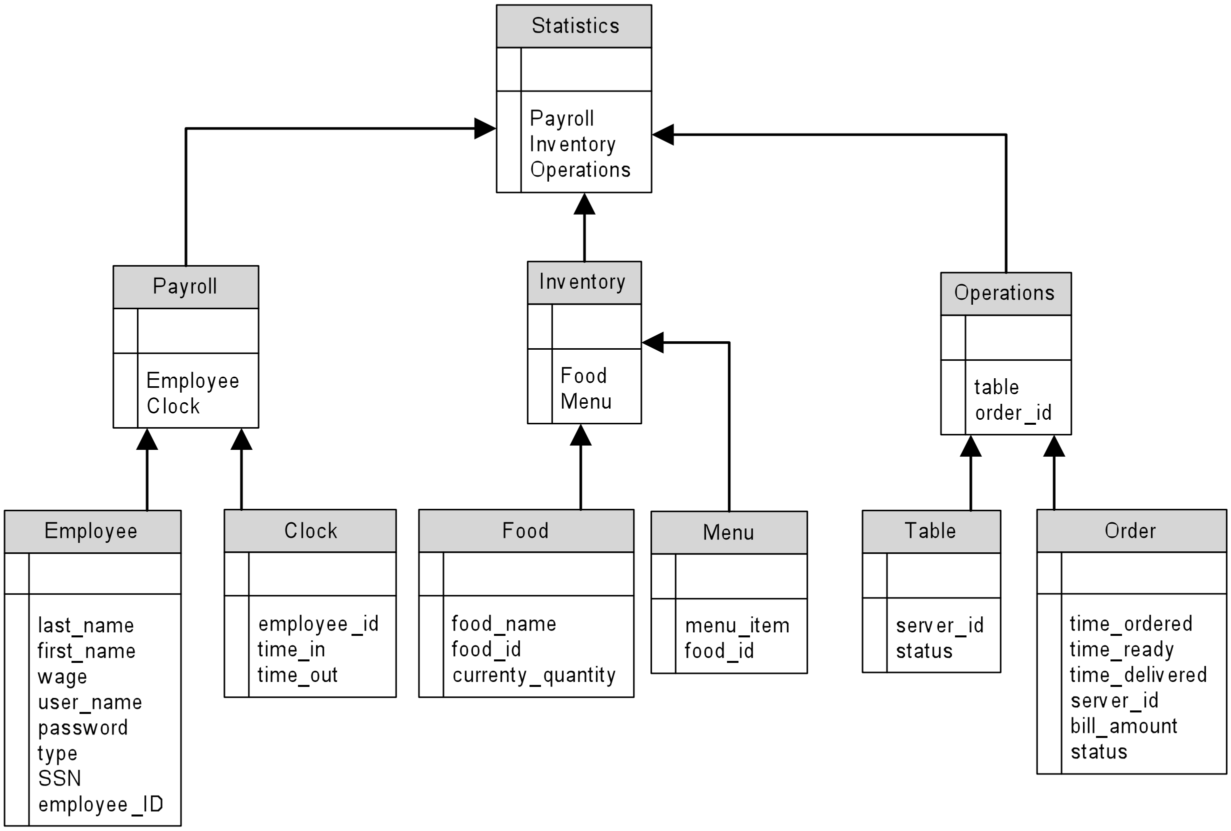


Figure 8: Diagram showing how data is stored in the system.

The overall database system will have three different categories: payroll, inventory, and operations. Each of these categories will allow for overall queries that support the functionality for summary financial, inventory, operational, and payroll management. Below is a description of the persistent data objects:

- Employee: stores information about each particular employee. Each employee can then be identified by the system for payroll, work assignment, etc.
- Clock: facilitates payroll calculation.
- Food: stores a list of each food item that the restaurant keeps on site and provides each item with an ID number. Orders, the menu, and inventory management will be driven by the food items in stock and their quantities.
- Menu: keeps track of which particular food items are contained in each order and how much of each is used in an order. This way, the inventory system can automatically update when a customer orders a particular menu item.
- Table: keeps track of the status of a table.
- Order: keep track of everything involved with a particular table's order.

E. NETWORK PROTOCOL

Multiple systems will communicate to a unified server; where all of the information will be stored and the web client will be hosted.

The web client will make use of an apache web server, which will make use of the HTTP web protocol. HTTP will be used because it falls under the client-server computing model, which allows for simple access to the web-based system from any device that has HTTP compatibility; which is almost universal for most devices. HTTP also allows for secure SSL encrypted connections via HTTPS, which allows for sensitive data to be encrypted and sent to a client with a smaller chance of a malicious program such as a listener being able to tap into the transmitting information.

The desktop java application and the java based android application will utilize JDBC (Java Database Connectivity). This protocol will be used because it supports execution and creation of SQL database statements. JDBC allows for simple integration of SQL database coding with streamline java coding, which will result in quick and efficient database queries and full compatibility with the software system.

Every system will require IPv4 as the internet protocol because it is currently universal and will allow for simple address ranges and domain mappings. The TCP/IP Model is going to be used in our system because it is also universal and is integrated within our system.

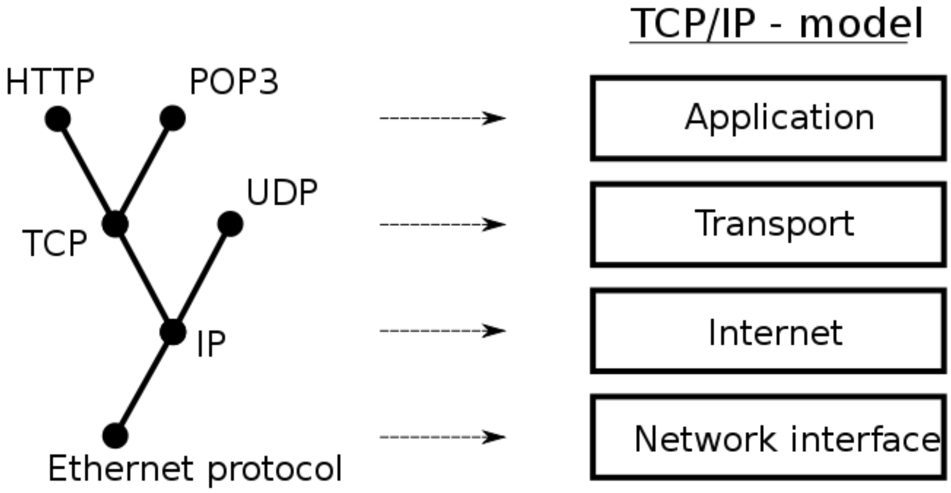


Figure 9

F. GLOBAL CONTROL FLOW

Execution Orderness

Our system will follow the event-driven model control flow. If no action is taken, the system will remain idle in a loop until a user initiates an event. Once a user causes an event through an action, a certain procedure will be followed until the task is finished. At this point, the system will again remain in the idle state, ready for more actions. With the immensity of a restaurant automation system, actions invoked by multiple users will be processed alongside other actions and events. This aspect is discussed below under concurrency.

Time Dependency

Our system contains timers mainly to keep track of restaurant statistics. These timers exist in the clock in/clock out of all restaurant personnel to record the amount of time worked and as a result, used in the calculation of payroll. In addition, the system includes a timer for the customer turnover, allowing for statistics on customer wait times and a general overview of the restaurant efficiency.

Since the uses for the timers record the amount of time that passes in accordance to the amount of time that passes in the real world, our system is considered a real-time system.

Even when no events occur, the system continues to count the amount of time employees are working. The only periodicity within this system as a real-time simulation is the maximum number of working hours per day, which repeats for each day.

Concurrency

Since our system requires information to be accessed and changed at the same time as due to the actions of several users, multiple threads are needed. The customers and waiters can place orders at the same time, resulting in the need for concurrency. This situation is taken care of by running the threads through a queue; storing and displaying all the orders for the chef. Also, during this time, the manager may be checking the restaurant and stock statistics. In order to take this situation into account, the system will give precedence to the manager before changing the data to allow the manager to see all statistics at the time the manager requested to view the stats. As a result, the cross between the commitment and timestamp ordering methods are used to control the concurrency.

G. MINIMUM HARDWARE REQUIREMENTS

Server

Our application will be using a server for the database. We plan on using a SQL database so we will make our hardware requirements as such. The following hardware requirements are from Microsoft's (owners of SQL) website.

Hardware	**Minimum Requirement**
Processor	1 GHz
RAM	512 MB
Hard Drive Space	3.6 GB
Network	10/100/1000 NIC Wifi 802.11n

Table 1.1: The minimum hardware requirements for the server.

Hand Held Devices

The application will make use of handheld Android phones and tablets. There is a wide variety of different devices that an Android program can run on. These devices all have different hardware. For our program, we will have a specific criterion for the hardware requirements. To increase usability, productivity, and hardware longevity, we specifically need handheld hardware that will allow our program to run without any lag or sluggish feeling. Also, the hand held devices should have a screen that is big enough to be seen easily and that has a relatively high screen resolution.

Hardware	**Minimum Requirement**
Processor	1 GHz dual core
RAM	1 GB
Hard Drive Space	16 GB (most newer devices come with this standard)
Network	4g (HSDP+, WiMax, 4gLTE) WIFI 802.11n
Screen	size: 4.0’’ Resolution: WVGA(480x800)

Table 1.2: The minimum hardware requirements for hand held devices

Desktop Client

In addition to a server and hand held device, we will also implement a desktop client. The minimum requirements for our desktop will not be as critical as for the handheld device because we will just be running a java application. However, we should have a big enough screen so all employees can view the screen easily.

Hardware	**Minimum Requirement**
Processor	1 GHz
RAM	512 MB
Hard Drive Space	5 GB
Network	10/100/1000 NIC Wifi 802.11n
Screen	Size: 20”

	Resolution: VGA(640x480)

Table 1.3: The minimum hardware requirements for the desktop client.

X. ALGORITHMS AND DATA STRUCTURES

A. ALGORITHMS

The algorithms used in the application are fairly simple. However, to make the process of implementing said algorithms straight forward, they must be clearly stated to avoid possible errors in the programming stage of development. There are two main tasks that require the use of algorithms: Check Stats/Stock and Edit Layout. The algorithms will be implemented as follows:

Check Stats/Stock

- Check Stats pertains to the checking of the restaurant statistics, which are menu trends (how well and item sells over a given period of time) and customer trends (how many customers attend the restaurant at various times over a given time period).

  Menu trends require the user (manager) to input a given time range and (optionally) a given item(s). The request is checked to be valid, and if it is, the process continues. Whenever an item is ordered, the stats database value under the current date and under the subcategory of the specific item sold is incremented, so the system simply queries the number under the category of the dates given and under the category of the specified items. Once this data is retrieved, the information is plotted on a line graph, the x axis being time (i.e. days of the week) and the y axis being the number of units sold of said item(s) (i.e. hamburger, salad, etc.).

  Customer trends are a very similar process. When a party of customers is seated the stats database value under the current date and under the subcategory of the current hour is increased by the number of customers in the seated party. The system retrieves this information for a given time range and plots the data on a line graph, the x axis corresponding to time and the y axis corresponding to number of customers. The customer trends is given greater precision than menu trends (hours vs. days, respectively) because it is more important to know specifically what times of the day are busier than others. It is important to know how many items are sold in a given day, but the extra precision (hours instead of days) is unnecessary and only complicates the process.
- Check Stock simply retrieves the current amount of each item in the stock database and displays it as a bar graph. The x axis shows ingredients and the y axis show amount.

Edit Layout

Edit layout is the algorithm for rendering the tables when the user (manager) visually edits the restaurant layout. Each table object saves a value for its x and y coordinate. The system renders the table on the screen at the given coordinates. When the user moves the table, the x and y coordinates of the new placement will be received. These new values are first verified. The system checks to see if the coordinates are within a given radius of another table. The radius will depend on the size and orientation of the table. For a round table, the radius is simply the table radius plus a defined amount to account for seating. For a square table, the radius will be sqrt(2)/2*width plus the defined distance for seating. A rectangular table will simply be a combination of square tables that uses the individual validity checking of each table to see if its placement is valid. However, to avoid seeing itself as an obstruction, the table object will be given a reference to the table objects it is combined with so it knows to ignore them when checking if its placement is valid.

B. DATA STRUCTURES

Our system focuses on two main types of data structures in particular: the queue and the array.

The queue data structure will be used for two particular situations. The first situation would be customers waiting to be seated. When a party enters the door of the restaurant, that particular group would be added to the queue. Consecutive parties that arrive after would be added to the queue. The queue works as a first in first out data structure meaning that the first elements to enter the queue will be the first to exit the queue. Such a structure is appropriate for customers waiting to be seated since the first to arrive would be seated first. The queue offers the best performance for this situation since it is quick in regard to adding and removing elements. Our queue would be a set size due to safety standards in which only a certain amount of people would be allowed to wait in the front of the house. Thus, the characteristic of the queue of having a limited size is not an issue. The second situation is for receiving orders, the requests are put into a queue, then returned in the order they were placed into the queue. With this setup, the first person to order is the first to be served. However, the chef is given view of all orders simultaneously. The queue simply allows these orders to be properly displayed in the sequence they came in. At Applebee's, the kitchen has a computer screen in which the orders are displayed in the sequence they came in. Once the order is completed, a person on the GU line prints a ticket and places the ticket on the plate. The order is now taken off the screen. The exact sequence in which the order was placed is often not followed at Applebee's, but in our restaurant we will be following the precise sequence the order came in.

The array data structure is useful for holding the status of all the tables in the restaurant. A disadvantage often cited for arrays is that the size of an array is static, but with the restaurant having a constant number of tables with each table being capable of holding a maximum amount of guests, this disadvantage is not an issue. The array structure will hold all the tables. Each table will be a data element with information such as the table status (clean, dirty, or occupied), the maximum amount of people that could be sat at that table, the bill that is to be assigned to that table, and the server to be assigned to that table. There will be various table sizes in the restaurant. Specifically, our restaurant will be divided into sections with each section being assigned to a particular server. There will be three section formats: one for the AM shift, one for PM shift for Monday to Thursday, and one for the PM shift for Friday to Sunday. The restaurant will consist of 15 tables that could accommodate 4, 11 booths that could accommodate 6, 6 booths that could accommodate 4, 1 booth that could accommodate 8, 1 table that could accommodate 6, and 8 tables that could accommodate 2. The specifics have been taken directly from the Applebee's setup. Details as to how this information was acquired are mentioned in the Interview section. Additionally, when a large party consisting of more than could be accommodated at a single table or booth would require tables toe be moved accordingly. With an array structure, we could loop through the entire array and see if there are contiguous tables where the sum of the maximum amount of people that could be sat is equal or one to two less than the party size. If we were to place two tables together to represent one party, we can simply have these two tables be marked as occupied in the array as well as have both tables be allocated to the same bill.

For seating customers, the request to be seated is taken by the hostess. However, due to the various sizes of groups, it wouldn't make sense to have to wait for a large group of customers to be seated when there is a small group that can currently be seated. In order to maximize customer throughput this situation must be avoided. The application solves this problem by using an array of queues to handle seating requests. The array holds a predefined number of queues corresponding to the number of customers in a group. The number of queues is determined by the maximum group size defined by the restaurant. For example, say a group of five people and a group of ten people request to be seated. The group of five is entered into the queue in the array corresponding to five and the same is done for the group of ten with their respective queue. When a table opens for five people, the system checks the corresponding queue in the array and notifies the hostess a table is available. The same is done when a table for ten is available. Combing these two data structures allows the application to properly seat customers.

XI. DESIGN OF TESTS

A. TEST CASES

Table 2.1: AddOrder

Test-case Identifier:	TC-2
Use Case Tested:	UC-2, main success scenario
Pass/fail Criteria:	The test passes if the user is successfully able to place an order where the stock in the database shows equal to or greater quantities of ingredients needed for the item.
Input Data:	Food Item
Test Procedure:	**Expected Result:**
Step 1. Select item to be ordered for which stock does not exist.	System displays ingredients and options of additions to order.
Step 2. Select extras to add to the order.	System displays the order and the extras requested.
Step 3. Submit order request.	System displays an error stating that the order cannot be placed due to a lack of ingredients and brings the user back to menu screen to choose items.
Step 4. Select item to be order for which stock exists.	System displays ingredients and options of additions to order.
Step 5. Select extras to add to the order for which stock exists.	System displays the order and the extras requested.
Step 6. Submit order request.	System displays a message stating that the order has been placed and returns the user to the menu.

Table 2.2: EditOrder

Test-case Identifier:	TC-3
Use Case Tested:	UC-3, main success scenario
Pass/fail Criteria:	The test passes if the user is successfully able to edit an order before the order has started being prepared.
Input Data:	Food Item

Test Procedure:	**Expected Result:**
Step 1. Select item that has been ordered and has started to be prepared.	System displays information about the item that was ordered.
Step 2. Select edit order.	System displays a message stating that the order has started to be prepared and can no longer be changed or canceled and returns user to menu.
Step 3. Select item that has been ordered and has not yet started to be prepared.	System displays information about the item that was ordered.
Step 4. Select edit order.	System displays a menu to change the items that were ordered.
Step 5. Select options and submit	System displays the changes to be made, prompts the user that the changes have been successfully made and returns the user to the menu.

Table 2.3: PayBill

Test-case Identifier:	TC-5
Use Case Tested:	UC-8, main success scenario
Pass/fail Criteria:	The test passes if the user is successfully able to complete a transaction with a credit card or pays the bill completely with cash.
Input Data:	Credit Card Information
Test Procedure:	**Expected Result:**
Step 1. Select Pay Bill.	System displays items ordered with prices and the total bill.
Step 2. Select credit as form of payment and enter invalid credit card information and submit.	System displays and error stating that transaction could not be completed because credit card information is invalid.
Step 3. Select credit as form of payment and enter valid credit card information for card with less credit than bill and submit.	System displays and error stating that transaction could not be completed because it was declined by the credit company.
Step 4. Select credit as form of payment and enter valid credit card information for card with credit greater than or equal to bill and submit.	System displays message stating that the bill was successfully paid.
Step 5. Select cash as form of payment.	System prompts user to wait to waiter.

Table 2.4: AddInventoryItem

Test-case Identifier:	TC-6
Use Case Tested:	UC-11, main success scenario
Pass/fail Criteria:	The test passes if the user is successfully able to add an item to the inventory
Input Data:	Inventory Item, Stock
Test Procedure:	**Expected Result:**
Step 1. Select manage inventory from main menu.	System displays a list of inventory items and the current stock associated with the items.
Step 2. Enter new item name and negative stock value.	System displays the entered information.
Step 3. Select add item.	System displays message stating that the item cannot be added because invalid stock value was entered.
Step 4. Enter new item name and positive stock value greater than 100000 units.	System displays the entered information.
Step 5. Select add item.	System displays message stating that the item cannot be added because invalid stock value was entered.
Step 6. Enter new item name and positive stock value less than 100000 units.	System displays the entered information.
Step 7. Select add item	System stores the new item and stock value in the database and displays a message stating the entry has been added to the database.

Table 2.11: AddInventory

Test-case Identifier:	TC-7
Use Case Tested:	UC-13, main success scenario
Pass/fail Criteria:	The test passes if the user is successfully able to add stock to an inventory item that exists in the database.
Input Data:	Inventory Item, Stock
Test Procedure:	**Expected Result:**
Step 8. Select existing inventory item.	System prompts user to add stock, remove stock, or cancel.
Step 9. Select add stock.	System prompts user to enter a value.
Step 10. Enter value greater than 100000 – current stock and select submit.	System prompts user that entered value is too large and prompts user to enter another value.
Step 11. Enter value less than or equal to 100000 – current stock and select submit.	System updates the stock value in the database for the item chosen and returns user to screen with inventory items.

Table 2.5: RemoveInventory

Test-case Identifier:	TC-8
Use Case Tested:	UC-14, main success scenario
Pass/fail Criteria:	The test passes if the user is successfully able to remove stock from an inventory item that exists in the database.
Input Data:	Inventory Item, Stock
Test Procedure:	**Expected Result:**
Step 12. Select existing inventory item.	System prompts user to add stock, remove stock, remove item, or cancel.
Step 13. Select remove stock.	System prompts user to enter a value.
Step 14. Enter value greater than current stock and select submit.	System prompts user that entered value is too large and prompts user to enter another value

Step 15. Enter value less than current stock and select submit.	System updates the stock value in the database for the item chosen and returns user to screen with inventory items.

Table 2.6: RemoveInventoryItem

Test-case Identifier:	TC-9
Use Case Tested:	UC-12, main success scenario
Pass/fail Criteria:	The test passes if the user is successfully able to remove an existing inventory item.
Input Data:	Inventory Item
Test Procedure:	**Expected Result:**

Table 2.7: UpdateEmployee

Test-case Identifier:	TC -12
Use Case Tested:	UC-22, main success scenario
Pass/fail Criteria:	The test passes if the user inputs valid employee information.
Input Data:	FirstName, LastName, Wage, Username, Password, Type, SSN
Test Procedure:	**Expected Result:**
Step 1. Replace old FirstName and type in an invalid new FirstName	System places a "×" next to the text field.
Step 2. Replace old FirstName with a valid new FirstName	System places a "✓" next to the text field.
Step 3. Replace old LastName with an invalid new LastName	System places a "×" next to the text field.
Step 4. Replace old LastName with an valid new LastName	System places a "✓" next to the text field.
Step 5. Wage is changed to an invalid input.	System places a "×" next to the text field.

Step 6. Wage is changed to a positive number.	System places a "✓" next to the text field.
Step 7. Replace Username with an invalid new Username.	System places a "×" next to the text field.
Step 8. Replace Username with a valid new Username	System places a "✓" next to the text field.
Step 9. Replace Password with an invalid new Password	System places a "×" next to the text field.
Step 10. Replace Password with a valid new password.	System places a "✓" next to the text field.
Step 11. Change Type to an invalid occupation.	System places a "×" next to the text field.
Step 12. Change Type to a valid occupation.	System places a "✓" next to the text field.
Step 13. Change SSN to an invalid number.	System places a "×" next to the text field.
Step 14. Change SSN to a valid number	System places a "✓" next to the text field.

Table 2.8: AddEmployee

Test-case Identifier:	TC -13
Use Case Tested:	UC-20, main success scenario
Pass/fail Criteria:	The test passes if the user adds an employee that does not already exist in the database.
Input Data:	Employee
Test Procedure:	**Expected Result:**
Step 1. Add an existing employee.	System prompts user that the employee already exists in the database.
Step 2. Add a new employee.	System adds the new employee's profile to the database; records successful addition of employee.

Table 2.9: RemoveEmployee

Test-case Identifier:	TC -14
Use Case Tested:	UC-21, main success scenario
Pass/fail Criteria:	The test passes if the user deletes an employee already existing in the database.
Input Data:	Employee
Test Procedure:	**Expected Result:**
Step 1. Delete an employee not currently in the database	System prompts user that the employee is not in the system.
Step 2. Delete an employee currently in the database.	System removes employee from database; records successful employee termination.

Table 2.10: AddMenuItem

Test-case Identifier:	TC -15
Use Case Tested:	UC-16, main success scenario
Pass/fail Criteria:	The test passes if the user enters a valid food item that is not already on the menu.
Input Data:	FoodItem
Test Procedure:	**Expected Result:**
Step 1. Type a FoodItem that already exists on the menu and click Add	System prompts user that the item is already on the menu.
Step 2. Type a FoodItem that does not already exist on the menu and click Add	System adds item to the menu; records successful addition of food item.

Table 2.11: RemoveMenuItem

Test-case Identifier:	TC -16
Use Case Tested:	UC-17, main success scenario
Pass/fail Criteria:	The test passes if the user enters a valid food item that is an existing menu item.
Input Data:	FoodItem
Test Procedure:	**Expected Result:**
Step 1. Type a FoodItem the does not already exist on the menu and click Delete	System prompts user that the item is not on the menu and therefore cannot be deleted.
Step 2. Type a FoodItem that is currently on the menu and click Delete	System deletes the specified item from the menu; records successful addition of food item.

B. UNIT TESTS

1. Table

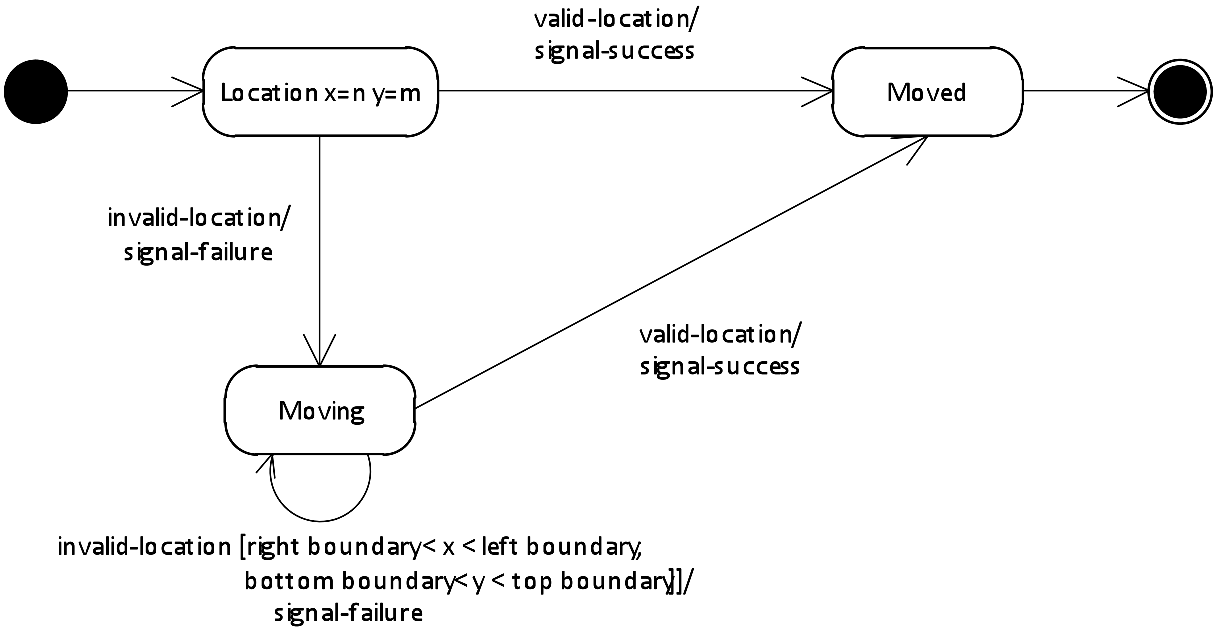


Figure 10.1: State diagram of the Table Class.

Method calls to test all states and transitions

```
Table A, B; // First tables are made.

editTableLayout(B.location(21,32));
editTableLayout(A.location(-1,-1));
editTableLayout(A.location(-1,15));
editTableLayout(A.location(15,-1));
editTableLayout(A.location(15,30));
```

The first method tests a valid location from the beginning. This will test the transition from the initial state to the final state of moved. The methods afterward, done on table A, take all possible invalid locations which are outside the boundary of the restaurant. This represents the transition from the initial location of a table to the state of moving with a notification that the table cannot be moved to the new location specified. On the last method, a valid location is inputted, showing the transition from the state of moving to the state of Moved. The system then notifies the user of the successful relocation of a table.

2. FoodProgress

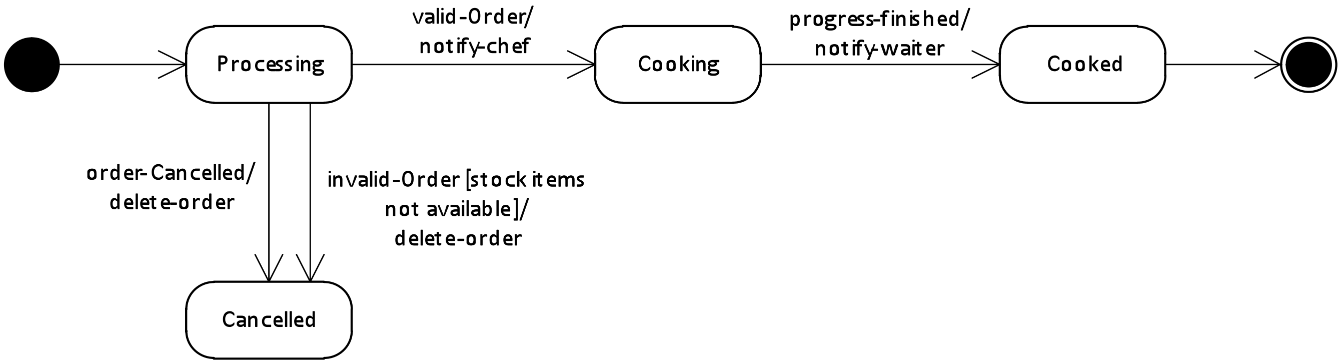


Figure 10.2: State diagram of the FoodProgress Class.

```
Method calls to test all states and transitions
Order O, F, N; //First an order is made
O.FoodItem(hamburger);
F.FoodItem(steak);
N.FoodItem(hotdog);

invalidOrder(O);
        //expected output True
invalidOrder(F);
        //expected output False; program automatically calls cancelOrder(F);
invalidOrder(N);
        //with no hotdogs in the inventory, this order is cancelled.
cancelOrder(O);
addOrder(O);
setOrderStatus(Cooking);
setOrderStatus(Finished);
```

First, all cancellations from a processing order are tested. This is done with the first methods calls with invalidOrder. If the method returns true, then the order is valid. However, if the ordered item is not a foodItem on the menu or if there is not enough stocks in the inventory to make the ordered food, the order will be cancelled by the system and the waiter notified about the situation. Also, if the customer decided to cancel an order, it will be allowed, since cooking the order has not been started.

Once a foodItem is processed, the Order is added to the queue of foods for the chef to cook. Once the chef gets to the specified food item, he/she changes the status of progress to cooking. This is seen in the transition from processing to the state of cooking. When the chef is done cooking, the order status is set to finished, representing the final state of cooked.

3. DBConnection

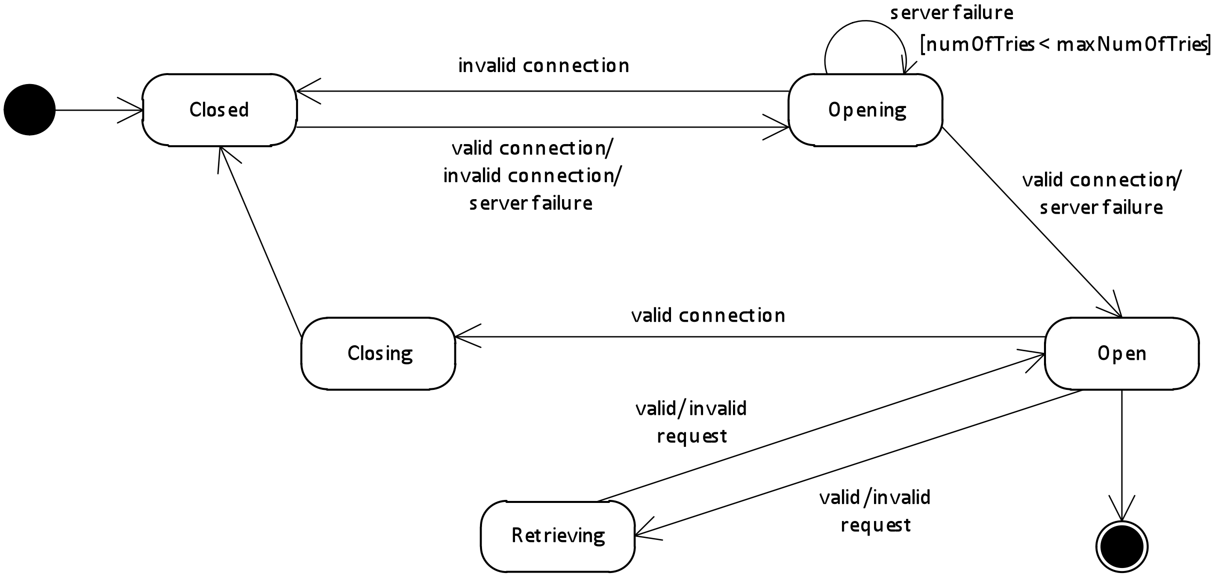


Figure 10.3: State diagram of the DBConnection class.

DBConnection.openConnection(); //Called to open a connection
DBConnection.closeConnection(); //Called to close a connection
DBconnection.queryDatabase(q); //Called to send a query, q, to the database

Initially, the state of the database connection is closed. The first test is connecting to the database, which is transitioning from the state closed to open. The method openConnection is called in order to do this. When this method is called the state goes to Opening. If the connection is invalid, the method returns from the state Opening to Closed. If there is a failure contacting the server, the method will allow maxNumOfTries tries to establish a connection and if a connection is established the state will be Open, otherwise, the state will be Closed. Once a connection is established requesting a query is tested with the method queryDatabase(q), which is going from the state Open to Retrieving and back to Open. Whether a valid or invalid query is requested, the state changes occur because querying a MySQL database will return the request no matter the request. Then closing the connection is tested using the method closeConnection(), which will move the state from Open to Closing to Closed. There are no alternate scenarios in closing a connection because JDBC (Java Database Connection) allows for safe closing of connections and the server closes a connection after a certain time of inactivity. As a result, the state will always go from Closing to Closed.

4. Employee

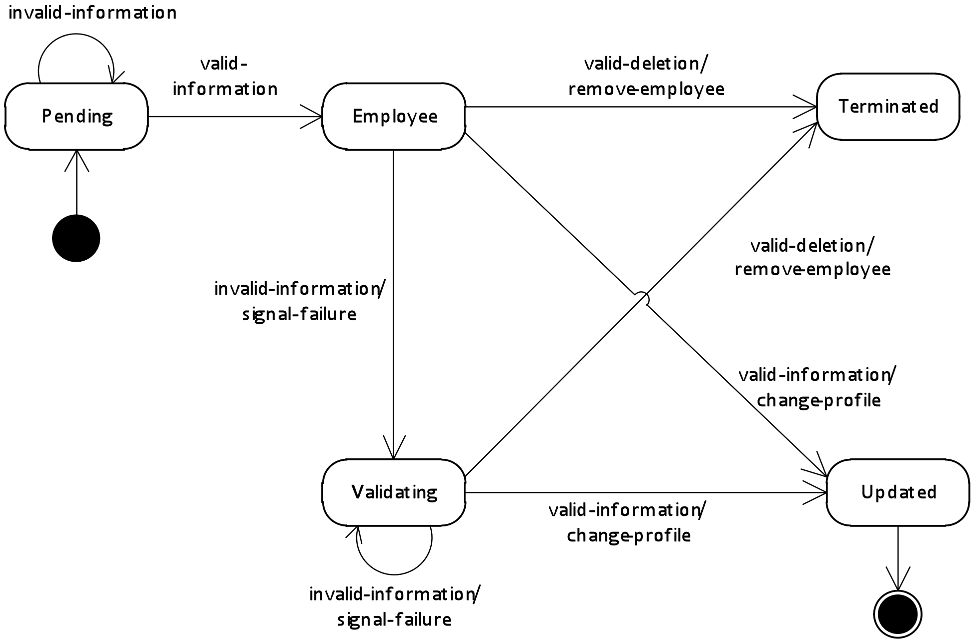

invalid-information
valid-information
Pending
Employee
valid-deletion/ remove-employee
Terminated
valid-deletion/ remove-employee
invalid-information/ signal-failure
valid-information/ change-profile
Validating
valid-information/ change-profile
Updated
invalid-information/ signal-failure

Figure 10.4: State diagram of the Employee class.

```
Employee E; //Create a valid employee
Employee N; //Create an invalid employee

addEmployee(N); //expected output False; invalid information
addEmployee(E); //expected output True;
editEmployee(N); //expected output False; employee does not exist
editEmployee(E); //expected output True;
removeEmployee(N); //expected output False; employee does not exist
removeEmployee(E); //expected output True;
```

Initially the state of the employee is Pending because the employee is not yet in the system. The method addEmployee(N) is called to test adding an employee with invalid information and should return false because the employee cannot be added. Then the method addEmployee(E) is called and the state goes from Pending to Employee because the employee is successfully added to the system. Then the method editEmployee(N) is tested in order to test editing the employee and the state transition from Employee to Validating to Updated. Since the information is not valid the transition would go to validating and the request would time out. Then the method editEmployee(E) is called and since the information is valid the transition is from Employee to Updated. If there is a failure in signal, the state remains in Validating and then times out; however, if a signal is established the state transitions to Updated. Then removing an employee is tested using the method removeEmployee(N). The state transition is from Employee to Validating; however, the request times out because the information is invalid and the state is back to Employee. Then the method removeEmployee(E) is called which transitions from the state Employee to Terminated. The state can go from Employee to Validating if there is a signal failure and if a valid connection is established the state proceeds to Terminated.

C. INTEGRATION TESTING

The integration testing method to be used will follow the horizontal integration testing strategy of bottom-up integration. Since our system contains many lower-level components put together by controllers, bottom-up integration is suitable. By testing with the lowest levels of the hierarchy, each unit can be tested separately since they do not depend on each other. Once these "leaf" classes are tested, the testing continues to the next level of the hierarchy, including all classes that contain the "leaf" classes. These navigable classes are tested with the lowest units.

Also, bottom-up integration reduces the need for test stubs and drivers saving time and possible errors. If there is a error in a higher-level class, the bottom-up integration method allows for locating the error more easily. Therefore, the final system will be integrated with uniformity, unlike the vertical integration testing strategy, where each subsystem corresponding to different user stories is made separately.

XII. HISTORY OF WORK, CURRENT STATUS, AND FUTURE WORK

At the beginning of this project, our team had high expectations in terms of what we wished accomplish. And as the project progressed, we realized that not everything we wished to complete could be completed within the time frame due to the reports. Our group learned that before implementing, we would have to go through steps such as creating use cases as well as requirements. Our goal became more realistic over time, but we kept our initial goals in mind. Tasks were prioritized over the semester in terms of what was deemed important. Understanding the concepts taught in class was an essential part of constructing the report. Actual coding only took place only once we had finished Report One. The milestones and deadlines were not set to be strict but to be reminders that we had a certain amount of days to accomplish a task. As the project continued as well as other course work, dates were often edited. The plan of work from the 1st report to the 2nd report varies slightly in dates. Below is the final Plan of Work that was followed.

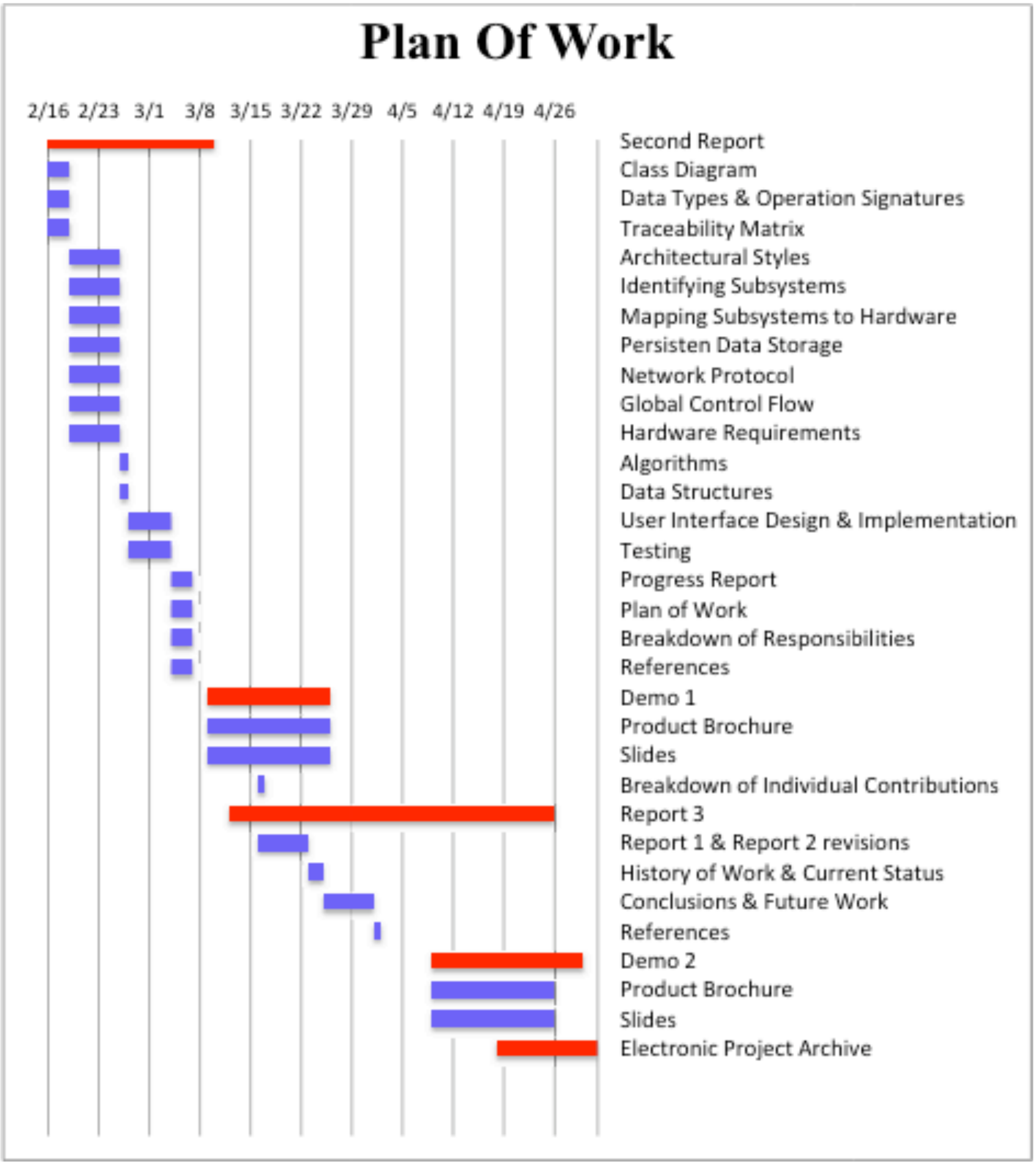
In terms of accomplishments, listed below are some of our groups:

- Applying real world situations to our system.
- Credit Card Reader that is iPhone compatible for customers to Pay Bills.
- Applying Software Engineering techniques to create a professional application.
- Creating a status of each individual table in the restaurant.
- Updating the customer continuously on the status of their order.

For future work, some features we would wish to implement are furthering the progress of viewing the current status of the tables in the restaurant. Given more time, we would hope that when a hostess wishes to view the status of the tables, a layout of the tables in the restaurant will appear. Depending on the state of the table (occupied, dirty, clean), each table will be a different color. Also, if a large party were to come into the restaurant such as a party of 20 for example, tables would be able to be dragged on the system to connect and form one table. One paycheck would be associated with this one table. Also, certain tables in the restaurant would be assigned to particular waiters, similar to what occurs in restaurants today.

Additionally, a more detailed menu would be implemented such that the user had options for toppings or extras such as cheese, lettuce, onions, etc. An instant notification message system for the waiter could also be created with which the waiter would be alerted when the Chef has completed the order. Also, an interface specifically for the delivery boy could be implemented as well. Another feature would be a "Happy Birthday" button that would automatically variations of the Happy Birthday song to customers that are at the restaurant to celebrate their birthday.

Our team would also hope to create a messaging system between the mobile and desktop application. Also, an application dedicated solely for the delivery boy. That application would have things such as a GPS as well as current orders that needed to be delivered. We would also hope to make the mobile application Android compatible. Many of these ideas we had hoped to accomplish but fell short with time.



Plan Of Work

2/16
2/23
3/1
3/8
3/15
3/22
3/29
4/5
4/12
4/19
4/26
Second Report
Class Diagram
Data Types & Operation Signatures
Traceability Matrix
Architectural Styles
Identifying Subsystems
Mapping Subsystems to Hardware
Persisten Data Storage
Network Protocol
Global Control Flow
Hardware Requirements
Algorithms
Data Structures
User Interface Design & Implementation
Testing
Progress Report
Plan of Work
Breakdown of Responsibilities
References
Demo 1
Product Brochure
Slides
Breakdown of Individual Contributions
Report 3
Report 1 & Report 2 revisions
History of Work & Current Status
Conclusions & Future Work
References
Demo 2
Product Brochure
Slides
Electronic Project Archive

XIII. INTERVIEW QUESTIONS

An interview was conducted with a hostess, waiter, and a manager at Applebee's. The questions are listed below.

Hostess

1. How do you go about sitting a guest when they come in?
It really depends on whether or not it's busy. If it's not busy then we sit the guests at the best place according to the guest size. Each server that is working during that shift has a section so we also try to rotate servers the best we can. You never want to double seat a server.

2. How do guests make a reservation?
Applebee's actually does not take reservations. We call them 'call aheads.' You can call ahead of time and say the number of guests as well as the time that you would like the reservation. We make sure to let the guests know that we do not take reservations but we accept call aheads meaning that we do not guarantee seating although we do try.

3. Can a guest cancel a reservation?
Yes. A guest would just need to call and let us know that they want to cancel their reservation. Most of the time though when a party wants to cancel a reservation, they simply don't show up.

4. How do you determine the time a guest has to wait?
If the restaurant is completely full, we are told to begin at a wait time of 10 minutes. Every guest after that is an additional 5. Parties of lets say 25 would have a longer wait since tables would need to be turned to accommodate them.

Server

1. When would an order be taken off the bill or when would a guest get a meal for free?
An order is taken off if the server had put in the system the wrong order or if the order comes out incorrectly cooked. Some customers complain that their meal is cold so that would be another situation. And sometimes if the wait for their food is too long. If it is the morning shift and a customer is waiting an hour for a salad, and it isn't busy then that would be a valid reason for a meal to be comped.

Manager

1. When would an employee get taken off the system?
An employee is taken off the system automatically if the employee hasn't worked for a month or two. The system keeps asking us if we want to keep this employee in the system because their account has been inactive. Another reason would obviously be the employee being fired.

2. Why would an employee get fired?
An employee would get fired if they weren't following the policies of the company, which are outlined in the employee handbook. Also, if an employee does not show up to work and does not have proper coverage or medical note, that employee is automatically fired. Also, if an employee

is giving out free food or beverages without manager knowledge.

XIV. REFERENCES

1. "Concurrency Control." *Wikipedia*. Wikimedia Foundation, 03 Nov. 2012. Web. 12 Mar. 2012. <http://en.wikipedia.org/wiki/Concurrency_control>.

2. "Hypertext Transfer Protocol." *Wikipedia*. Wikimedia Foundation, 03 Nov. 2012. Web. 12 Mar. 2012. <http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol>.

3. Marsic, Ivan. *Software Engineering*. 2012. PDF.

4. "Java Database Connectivity." *Wikipedia*. Wikimedia Foundation, 03 Nov. 2012. Web. 12 Mar. 2012. <http://en.wikipedia.org/wiki/Java_Database_Connectivity>.

5. "Real-time Simulation." *Wikipedia*. Wikimedia Foundation, 21 Jan. 2012. Web. 12 Mar. 2012. <http://en.wikipedia.org/wiki/Real-time_Simulation>.