Concepts in Software Engineering
Group 15
EZ-Serve

http://www.elromenterprises.com/ezserve

4/15/2007

Simon Rudin
Christian Santiago
James Rosado
David Elrom

Statement of Requirements

As of this moment, the restaurant staff will be logging into the system via touch screen to complete their desired transaction. After the waiter logs into the system, they will be taken to a screen that will consist of all the tables currently assigned to them. These tables will be given certain stages depending on their color. The color red signifies a dirty table, green signifies that the table is unoccupied, and yellow signifies a table that is currently occupied. In the upper portion of the screen there will be a window which will allow the waiter to be advised of all completed orders that he/she has yet to update. From this screen the waiter will also be able to select a table and view their current tab, and the status of their order. Once the desired table is selected, the waiter will be able to add an additional item to the tab from a list of various food items which will be categorized in the window. For instance, if a customer places an order for a Steak, the waiter would log in, select the appropriate table, and select “Add Item.” From this screen they will be given three categories to select from: Appetizers, Entrée, and Desert. For the steak the waiter would select Entrée and than proceed to the beef category which he/she will than select the desired item. After adding the item they will be returned to the table screen where they are given the option of performing another task or log out. From there the item will be placed on a queue that will be displayed to the cook by use of another monitor that will be present in the kitchen to allow the waiter to continue servicing other tables without having to run to the kitchen to place the order. The queue system that will be used for the cooks will be a modified version of a FIFO (First in First Out) system to adjust to the priorities that will be given to the food items. For instance, if a table orders an appetizer and an entrée at the same time, we would create queue such that the order for

the appetizer will be fulfilled first and than after a certain period of time the entrée will than begin to be prepared.

The supported classes for an employee are: Host, Waiter, Cook, Busboy and Manager. The system will allow for direct communication between the Host and Waiter, Waiter and Cook, and also Busboy and Host. To prevent fraud or system manipulation each account will be given certain privileges determined by the type of employee. All employee screens will be updates frequently to allow the Host to be notified as soon as a table is ready or allow a waiter to be notified of a completed order. The Manager will have administrative control over employee accounts. For example, the ability to create or modify a profile, track the number of hours worked, and view graphs and data charts concerning the restaurant sales and inventory. The Manager will also be able to monitor the tables that employees are currently assigned to, and the tabs of those tables.

The devices that the waiter will be frequently using to input these orders will be input stations which will be conveniently placed throughout the store for easy and quick access. Although it would be ideal to have handheld devices for the waiters to input their orders, it would be an expensive solution for a minor problem. The positive result of having a station to input orders is that it limits the possibility of misplacing or dropping a handheld device that would cost the manager even more financial woes. Another device that was being considered was the use of computer stations at every table. But through field work, we found that many customers did not exactly fancy the idea of having them at their table. So for this simulation of our system we will assume to have stationary stations fixed throughout the restaurant.

This system will require that the waiters frequently visit the stationary terminal to log in, place their order, check on the status of recent orders placed, and than log out. At first this may seem very repetitive and tedious but these frequent visits would allow for the system to refresh itself and update the waiter on the next log in that an order that was recently placed is ready to be served. The only terminals that will not be logged into frequently will be the terminals the kitchen staff and host will be occupying. These terminals will be modified to refresh their screens to update the status of a table or order placed.

A design issue that was encountered was how the users will log into the systems. The options under review were touch screen, card reader and an RFID (Radio Frequency Identification) System. Through field work, we found that many of the current card reader systems in service have actually been a cause of frustration among users because as time progressed the system became congested with particles of food and dirt which caused the card reader to not recognize employees and as a result cause more wasted time trying to swipe into the system. This was a major concern for a card reading system, also with the possibility of losing or damaging the cards this option seems highly costly in the long run. In addition a new card would have to be created every time a new employee is hired at the restaurant. So the card reader failed to be a successful venture. The second option was an RFID system. This looked promising but could also cause great grief because during our field work we saw that many of the employees gathered around the terminal in which they would use their card to log in. This may cause unwanted results since multiple RFID cards would be present at a terminal during one instance. This system would also create similar financial problems which were discussed with the card

reader system due to the usage of cards. Our final and most accepted design was a touch screen system which most employees preferred. This system was simple and financially wise in the log run. Also creating profiles on the system is essentially free for new employees, instead of creating new ID cards for them.

The final interface issue is the floor plan designing process. Ideally, we'd have created a system which would allow for the manipulation of the tables with walls and bar in place. But given the time constraints we will be implementing a generic version. We will require the Manager to enter the dimensions of the table being used and than a window will appear with a grid for the seating area. Than at each element of the array the Manager will have an option of either placing a table or keeping it unoccupied. After the arrangement is selected the setup will be stored for use later.

The system will also have the capability to store Employee information and organize them. This will allow for the manager to easily process payroll since all employee hours will be logged. Revenue will be tracked daily, weekly and monthly to allow for further analysis by the Manager. All the information that is stored with this system will allow for a wide range of statistics to be calculated and graphs to be generated to help in the refinement of the restaurant business. All these features will be done automatically and able to be accessed by only the manager.

To help in the refinement of the system, the developers will also be cautious of the amount of clicks/keystrokes that are necessary to complete a desired task. This will help in reducing the time spent at the terminal for the waiter during the heavy traffic hours. In minimizing the amount of time spent at the terminal, functionality and security must not be compromised.

Glossary of Terms

1. **Employee** – any individual involved in the day to day activities of the restaurant, including cooks, bus boys, waiters, managers, and hosts.

2. **Manager** – subtype of employee with greater administrative authority, which can be summarized as the power to change the status of all other employee types, and access to restaurant statistics and floor plan.

3. **Waiter** – subtype of employee whose sole responsibility is to interact with the customer of an occupied table, taking and submitting the orders, and changing the status of the table from busy to dirty when the service is over.

4. **Cook –** subtype of employee whose responsibility is to cook the submitted orders by taking responsibility for the available orders and informing the system upon their completion.

5. **Bus Boy –** subtype of employee who has the sole responsibility to clean the tables and change their status from dirty to clean upon completion of this task.

6. **Host –** subtype of employee who greats the incoming customers, directs them to available tables, or to a waiting queue. This activity can be performed by both waiters and managers, but not by cooks or busy boys.

7. **Table –** physical object upon which the cooked food is presented to the customers, this object is the center of the waiter's universe as far as his existence within the system is concerned.

8. **Dirty** – a status indicator referring to the condition of the table entity, whence this entity is not occupied with servicing customers, but is not ready to be used for a new customer just yet, as it is in need of cleaning.

9. **Busy** – a status indicator referring to the condition of the table entity, whence this entity is in service to the customer and cannot be cleaned or assigned to a new customer.

10. **Ready** – a status indicator referring to the condition of the table entity, when this entity can be assigned to a new customer, or in other words is vacant and ready for service.

11. **Order –** submitted description of an item to be cooked and eventually delivered to the table from which it has been ordered. This entity is created by the waiter, and is modified by the cook.

12. **Cooking –** a status indicator referring to the condition of the order, meaning that the order is currently being cooked by a cook.

13. **Cooked –** a status indicator referring to the condition of the order, meaning that the order has been cooked and is ready to be served to the customer.

14. **Password –** a secrete string of characters assigned to every employee in order to securely login into the system. Together with the employee social security number, this entity makes up the username-password pair for the login procedure.

15. **Timestamp –** the UNIX representation of system time which we are going to utilize for time keeping purposes throughout the system.

16. **Wage –** hourly rate at which an employee is being compensated for his or her work within the framework of the restaurant automation system.

17. **Customer –** the individual who is being served by the restaurant automation system and the employees working within its framework.

Stakeholders

1. **Owner –** Whoever owns the restaurant, whether it is a family run operation or a chain such as Red Lobster.
2. **Employees**
    - Manager
    - Waiter
    - Cook
    - Host
    - Bus Boy
3. **Customers –** Whoever happens to eat at the restaurant, these can be families, couples, groups of friends, etc.
4. **FDA –** As far as compliance with the Food and Drug Administration's policies regarding public eateries is concerned.

Actors and Goals

ACTOR	ROLE, DESCRIPTION of GOAL, UC's
BUS BOY	Initiating, goal is to clean the table and change the table status from dirty to ready, UC-1, UC-11, UC-12.
HOST	Initiating, goal is to assign CUSTOMERS to the tables, and change the table status from ready to busy, UC-2, UC-11, UC-12.
WAITER	Initiating, goal is to take the orders and report a dirty table once CUSTOMERS have left. UC-5, UC-11, UC-12, UC-13.
CUSTOMER	Participating, goal is to order food and pay bill, UC-5.
COOK	Initiating, goal is to receive orders and cook food. This also includes changing the order status from busy to ready, UC-3, UC-4, UC-11, UC-12.
MANAGER	Initiating, goal is to access various statistics, view/edit employee payroll, and add/drop/modify employees, UC-6, UC-7, UC-8, UC-9, UC-10, UC-11, UC-12, and UC-14.

Casual Use Case Descriptions

USE CASE	Description
Clean Table	The BUS BOY cleans the table and reports the table as clean.
Occupy Table	The HOST assigns a table to a customer and table status is changed to busy.
Make Order	COOK receives order and cooks it.
Deliver Order	Order status is set to ready.
Make Payment	WAITER assigns the table dirty after the CUSTOMERS have left.
Take Order	WAITER takes the food order and reports it to the COOK.
Request Statistics	The MANAGER requests the restaurant statistics.
Payroll	The MANAGER requests the payroll of a employee and it is displayed.
Modify Employee	The MANAGER requests to modify the employee and the employees data is changed.
Create Employee	The MANAGER requests to add an employee and the new employee is added to the payroll.
Delete Employee	The MANAGER requests to remove an employee and the employee is then removed.
Login	An employee attempts to log in, but their credentials have to be verified.
Logout	An employee logs out of the system.
Manage Menu	MANAGER attempts to add and delete food items from the menu.

Detailed Use Case Descriptions

Use case name	**Clean Table (UC-1)**	
Entry condition	Table status is dirty	
Exit condition	Table status is clean	
Quality Requirements	None	
Primary Actor	BUS BOY	
Main Flow	**Step**	**Action**
	1	BUS BOY logs in **Include::Login**
	2	BUS BOY removes dirty plates and silverware
	3	BUS BOY cleans tabletop
	4	BUS BOY sets the table
	5	BUS BOY reports clean table

Use case name	**Occupy Table (UC-2)**	
Entry condition	Table status is ready	
Exit condition	Table status is busy	
Quality Requirements	Number of people must correspond to table size	
Primary Actor	HOST	
Main Flow	**Step**	**Action**
	1	HOST logs in **Include::Login**
	2	HOST performs table search
	3	HOST selects appropriate table
	4	HOST changes table status to busy
Extensions	**Step**	**Queue Customer**
	2.1	HOST performs table search
	2.2	HOST doesn't find suitable table
	2.3	HOST queues customer
	2.4	Number of customers on the Queue increments

Use case name	**Make Order (UC-3)**
Entry condition	Order available to be taken
Exit condition	Order status changed to taken
Quality Requirements	None
Primary Actor	COOK
Main Flow	**Step** **Action** 1 COOK logs in **Include::Login** 2 COOK finds order 3 COOK selects order 4 COOK changes order status to taken

Use case name	**Deliver Order (UC-4)**
Entry condition	Order is cooked
Exit condition	Status of order is changed to ready
Quality Requirements	No order takes longer than 40 min. to cook
Primary Actor	COOK
Main Flow	**Step** **Action** 1 COOK logs in **Include::Login** 2 COOK changes order status to ready 3 Update outstanding orders panel on WAITER terminal

Use case name	**Take Order (UC-5)**	
Entry condition	CUSTOMER request a food item	
Exit condition	Order for food is created	
Quality Requirements	CUSTOMER is satisfied	
Primary Actor	WAITER	
Secondary Actor	CUSTOMER	
Main Flow	**Step**	**Action**
	1	WAITER logs in **Include::Login**
	2	CUSTOMER requests food item
	3	WAITER selects customer table
	4	WAITER clicks on Add Item button
	5	WAITER chooses entry category
	6	WAITER chooses food category
	7	WAITER chooses food item
	8	WAITER submits the order by clicking Add Item button
Extensions	**Step**	**Item Unavailable**
	2.1	WAITER determines if existing item can be modified to accommodate the CUSTOMER request (This is not a computerized operation, left up to the waiters discretion)
	2.2	If the determination is positive the WAITER supplies modification and submits the order
	2.3	If not the WAITER informs CUSTOMER that their order can’t be fulfilled

Use case name	**Request Statistics (UC-6)**
Entry condition	None
Exit condition	Appropriate statistics are displayed
Quality Requirements	None
Primary Actor	MANAGER
Main Flow	**Step** **Action** 1 MANAGER logs in **Include::Login** 2 MANAGER clicks on Statistics button 3 Statistics screen is displayed to the manager (The statistics displayed are a set group of graphical data displays which simply show the statistics they represent, and cannot be interacted with in any way)

Use case name	**Payroll (UC-7)**
Entry condition	MANAGER request payroll data
Exit condition	None
Quality Requirements	None
Primary Actor	MANAGER
Main Flow	**Step** **Action** 1 MANAGER logs in **Include::Login** 2 MANAGER clicks on Payroll button 3 Initial unfiltered payroll screen is displayed 4 MANAGER selects payroll data filters based on first name, last name, salary range, and social security number 5 Manager clicks Filter button 6 Filtered data is displayed to the MANAGER in the payroll screen

Use case name	**Modify Employee (UC-8)**
Entry condition	MANAGER requests employee modification
Exit condition	Employee data is modified
Quality Requirements	None
Primary Actor	MANAGER
Main Flow	**Step** **Action** 1 MANAGER logs in **Include::Login** 2 MANAGER clicks on Edit Profile button 3 Search profile screen is displayed 4 MANAGE enters search criteria and clicks Search 5 Results screen is displayed 6 MANAGER selects employee to edit and clicks Edit button 7 Employee edit profile screen is displayed 8 MANAGE enters changes to the profile 9 MANAGER submits modified employee data by clicking the Submit button 10 Main MANAGER screen is displayed

Use case name	**Create New Employee (UC-9)**
Entry condition	MANAGER requests employee addition
Exit condition	Employee is added to staff
Quality Requirements	None
Primary Actor	MANAGER
Main Flow	**Step** **Action** 1 MANAGER logs in **Include::Login** 2 MANAGER enters in new employee data 3 MANAGER submits new employee 4 Employee is added to staff list and payroll

Use case name	**Delete Employee (UC-10)**
Entry condition	MANAGER requests employee deletion
Exit condition	Employee is deleted from staff
Quality Requirements	None
Primary Actor	MANAGER
Main Flow	**Step** **Action** 1 MANAGER logs in **Include::Login** 2 MANAGER searches employee profile 3 MANAGER submits deletion request 4 Employee is deleted from staff listing and payroll

Use case name	**Login (UC-11)**
Entry condition	Primary Actor attempts log in
Exit condition	Identity is verified
Quality Requirements	None
Primary Actor	{BUS BOY, COOK, WAITER, HOST, MANAGER}
Main Flow	**Step** **Action** 1 System checks primary actor's credentials 2 System loads appropriate interface
Extension	**Step** **Identity Check Failiure** 1.1 System fails to identify primary actor 1.2 System returns log-in screen

Use case name	**Logout (UC-12)**
Entry condition	Primary Actor attempts to log out
Exit condition	Primary Actor is logged out
Quality Requirements	None
Primary Actor	{BUS BOY, COOK, WAITER, HOST, MANAGER}
Main Flow	**Step** **Action** 1 Primary Actor clicks Logout 2 Log in screen is displayed

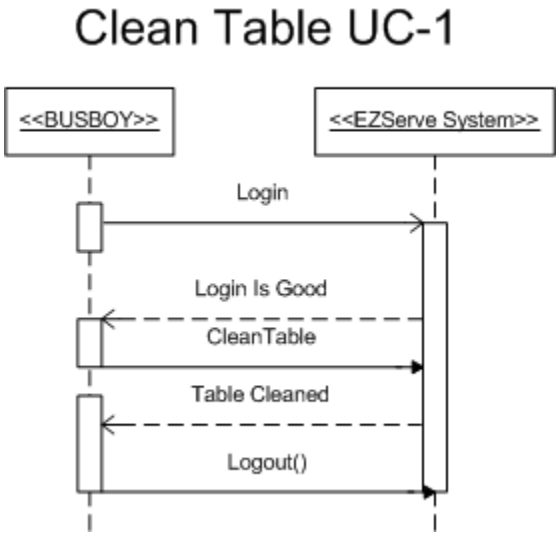
Use case name	**Make Payment (UC-13)**
Entry condition	WAITER attempts to close the tab
Exit condition	Table status is changed to dirty
Quality Requirements	None
Primary Actor	WAITER
Main Flow	**Step** **Action** 1 WAITER selects the table 2 WAITER clicks Close Tab 3 Table status is changed to dirty (Comment: this is essentially changing the table's status to dirty)

Use case name	**Manage Menu (UC-14)**
Entry condition	MANAGER attempts menu change
Exit condition	Menu is changed
Quality Requirements	None
Primary Actor	MANAGER
Main Flow	**Step** **Action** 1 MANAGER logs in **Include::Login** 2 MANAGER clicks on Menu button 3 Initial menu edit screen is displayed 4 MANAGER selects which type of entry to change 5 Entry edit screen is displayed 6 MANAGER can choose to delete an entry by clicking Delete Item button 7 MANAGER can choose to add an entry by clicking Add Entry

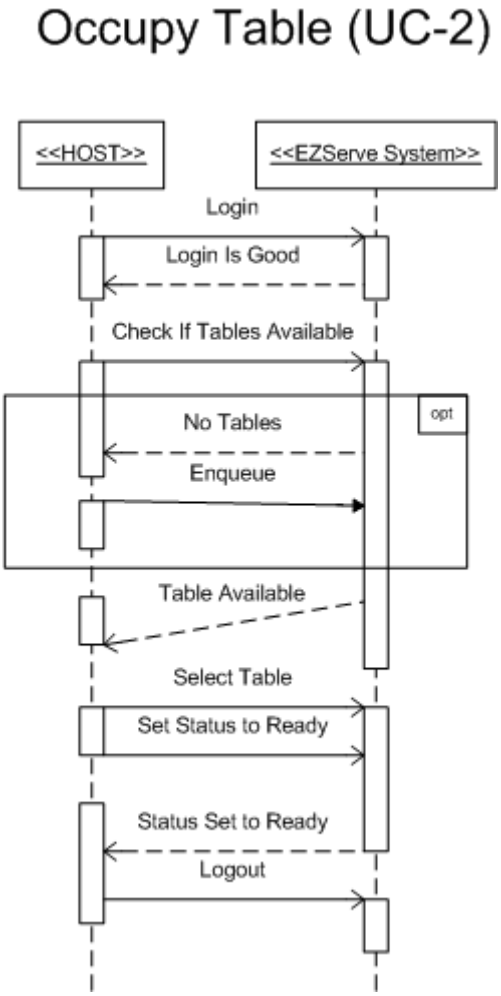
	8	MANAGER clicks submit to complete the change of menu operation
	9	Main MANAGER screen is displayed
Extension	**Step**	**Delete Item**
	6.1	Delete menu item screen is displayed
	6.2	MANAGER selects entry type
	6.3	Food types of selected entries are displayed
	6.4	MANAGER selects food type
	6.5	Foods of selected types are displayed
	6.6	MANAGER selects food and clicks delete
	6.7	Food item is deleted from the menu, and Menu Edit MANAGER screen is displayed
Extension	**Step**	**Add Item**
	7.1	Add item screen is displayed
	7.2	MANAGER entry type
	7.3	Food types of selected entrée type are displayed
	7.4	MANAGER selects food type
	7.5	MANAGER enters new food item
	7.6	MANAGER clicks Add Item
	7.7	Item is added to the menu and the Menu edit screen is displayed

System Sequence Diagrams

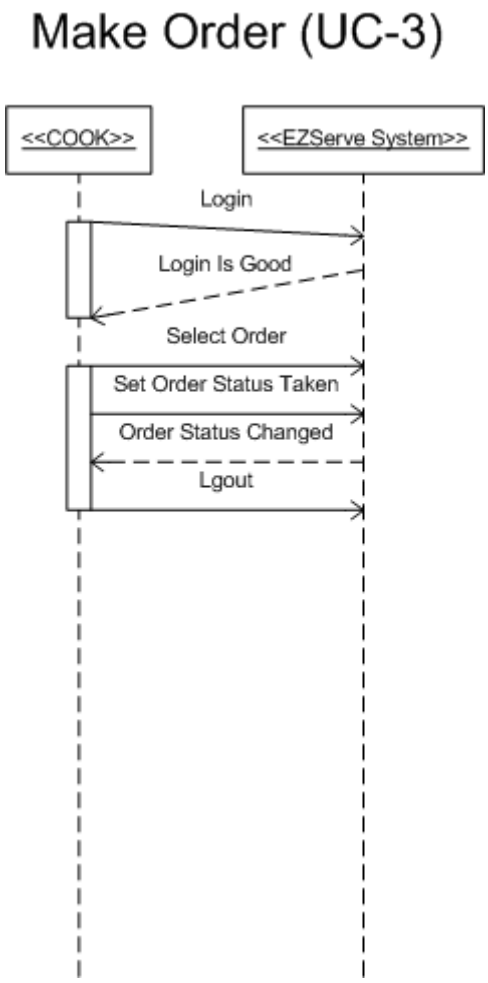
UC 1 (Clean Table)



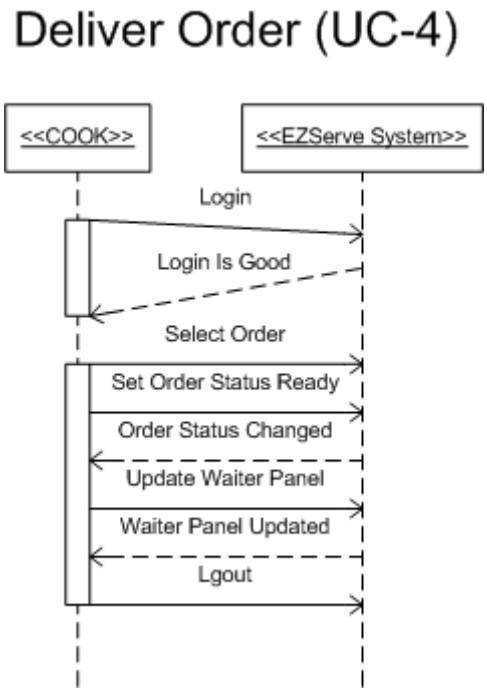
UC 2 (Occupy Table)



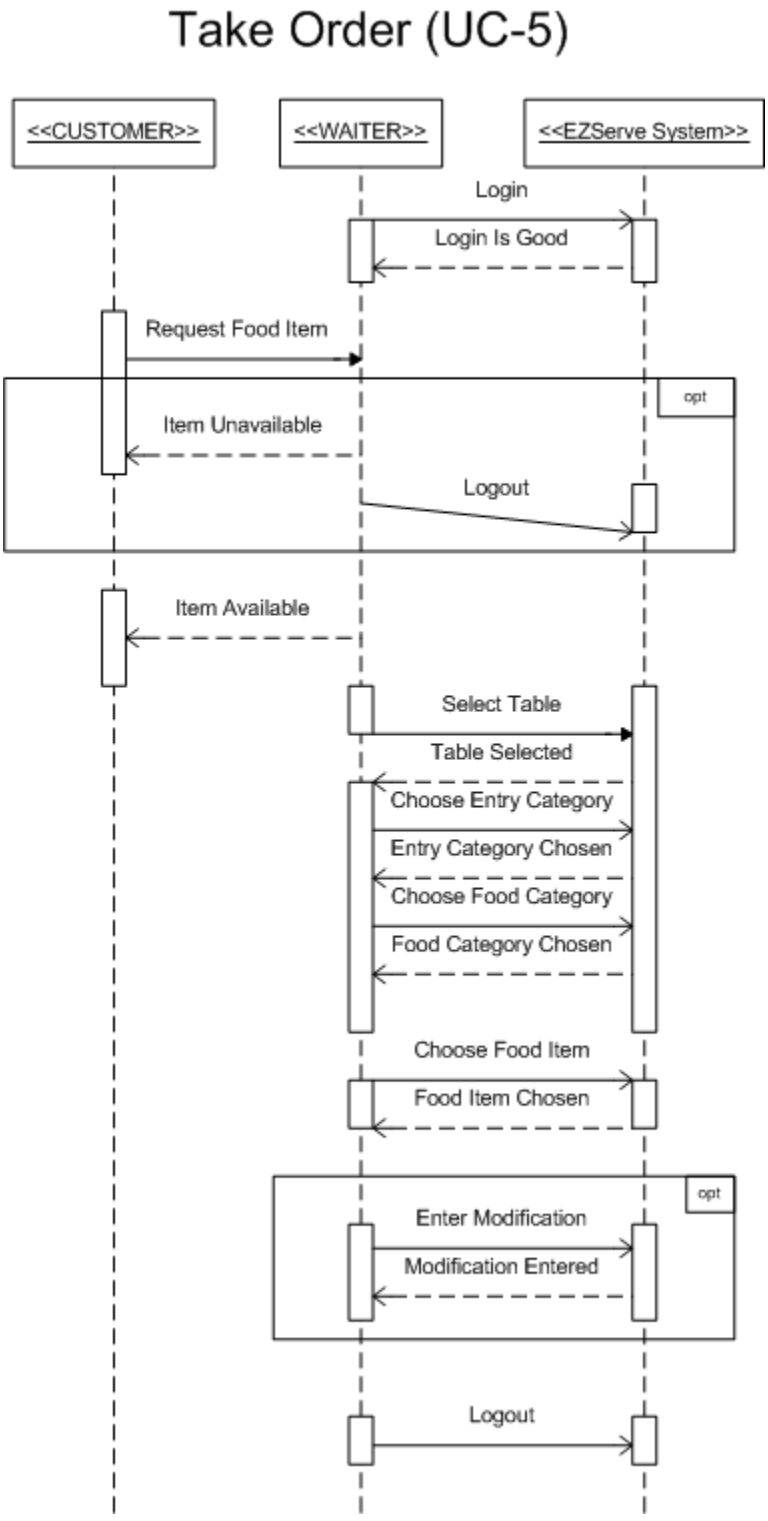
UC 3 (Make Order)



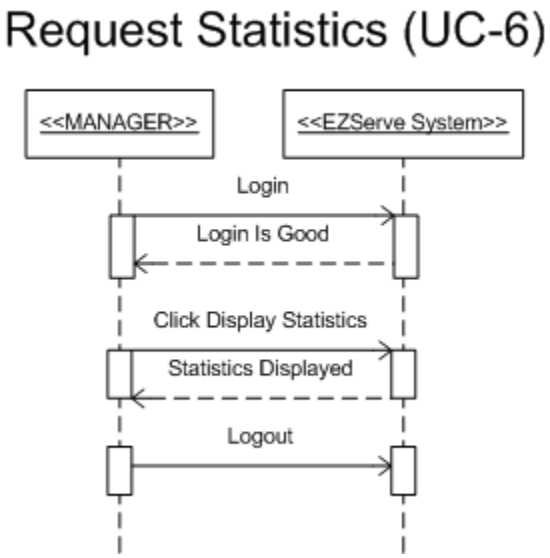
UC 4 (Deliver Order)



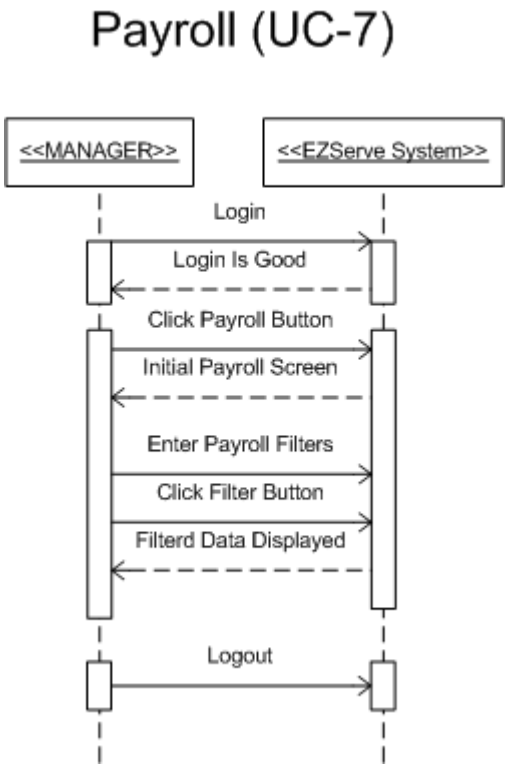
UC 5 (Take Order)



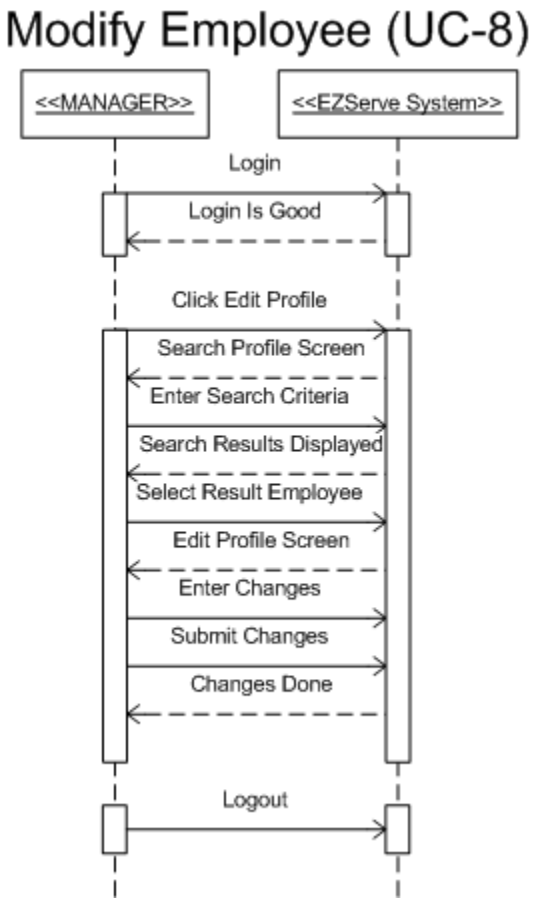
UC 6 (Request Statistics)



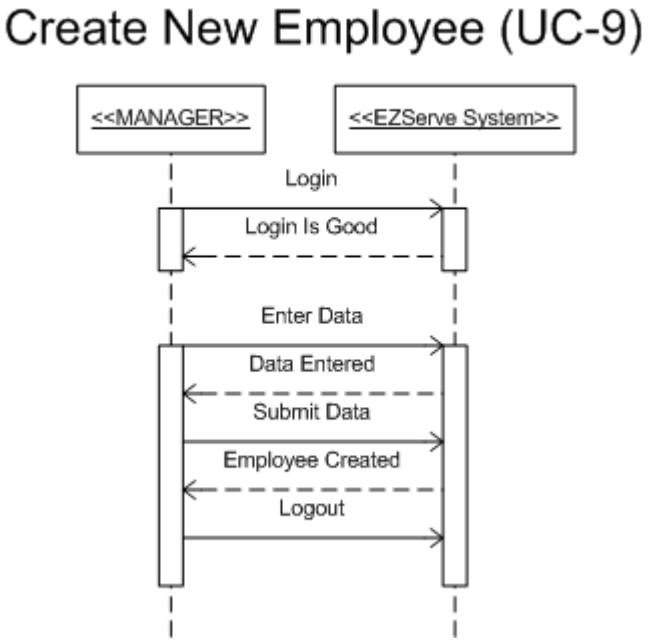
UC 7 (Payroll)


Payroll (UC-7)
<<MANAGER>>
<<EZServe System>>
Login
Login Is Good
Click Payroll Button
Initial Payroll Screen
Enter Payroll Filters
Click Filter Button
Filterd Data Displayed
Logout

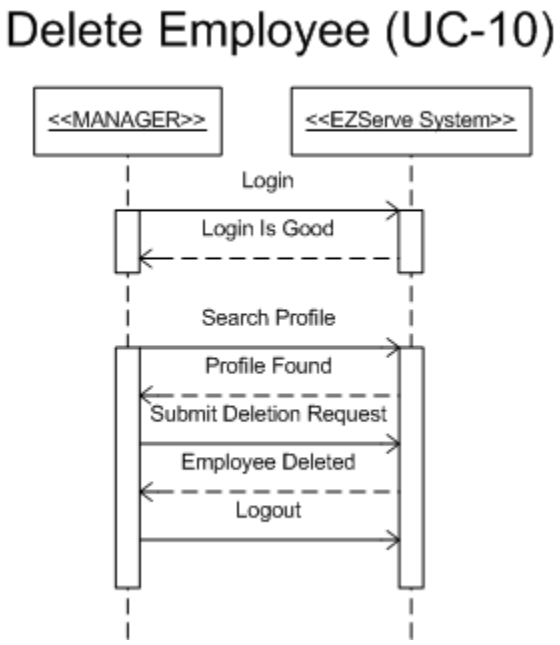
UC 8 (Modify Employee)



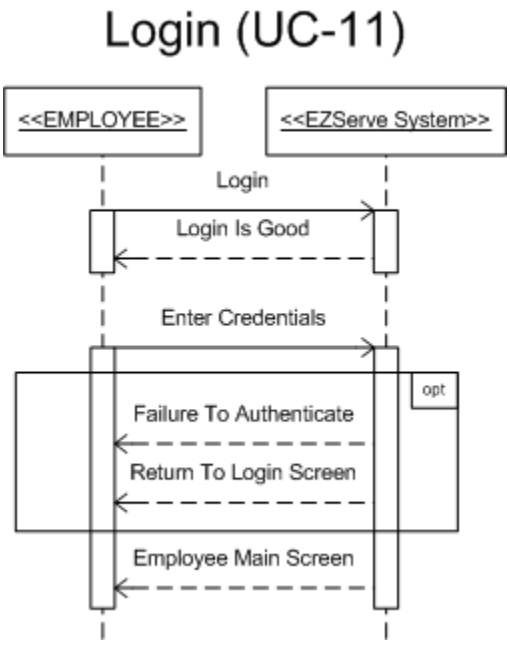
UC 9 (Create New Employee)



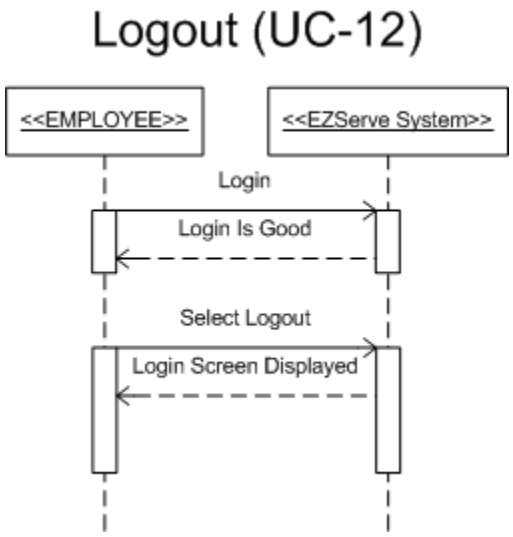
UC 10 (Delete Employee)



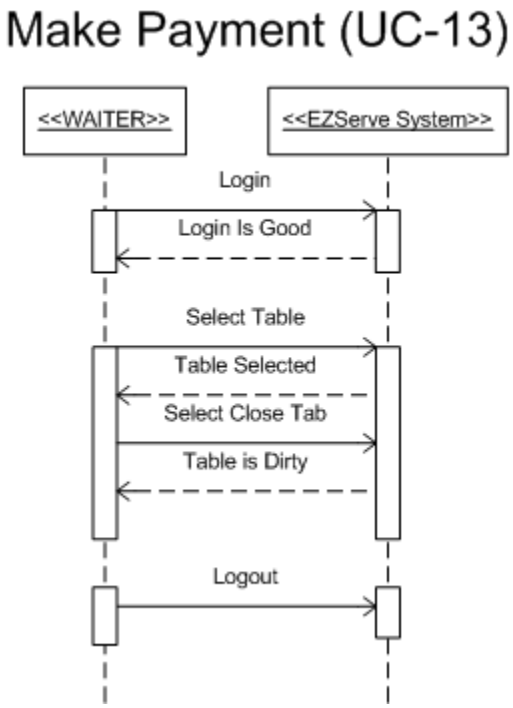
UC 11 (Login)


Login (UC-11)
<<EMPLOYEE>>
<<EZServe System>>
Login
Login Is Good
Enter Credentials
opt
Failure To Authenticate
Return To Login Screen
Employee Main Screen

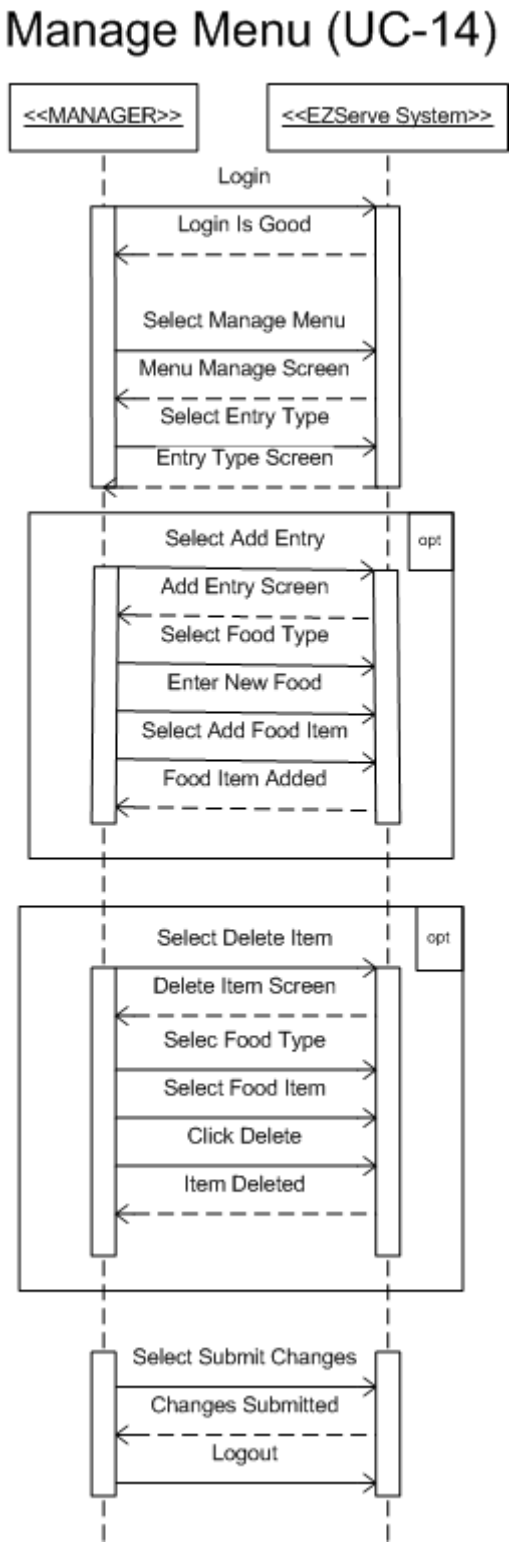
UC 12 (Logout)


Logout (UC-12)
<<EMPLOYEE>>
<<EZServe System>>
Login
Login Is Good
Select Logout
Login Screen Displayed

UC 13 (Make Payment)



UC 14 (Manage Menu)

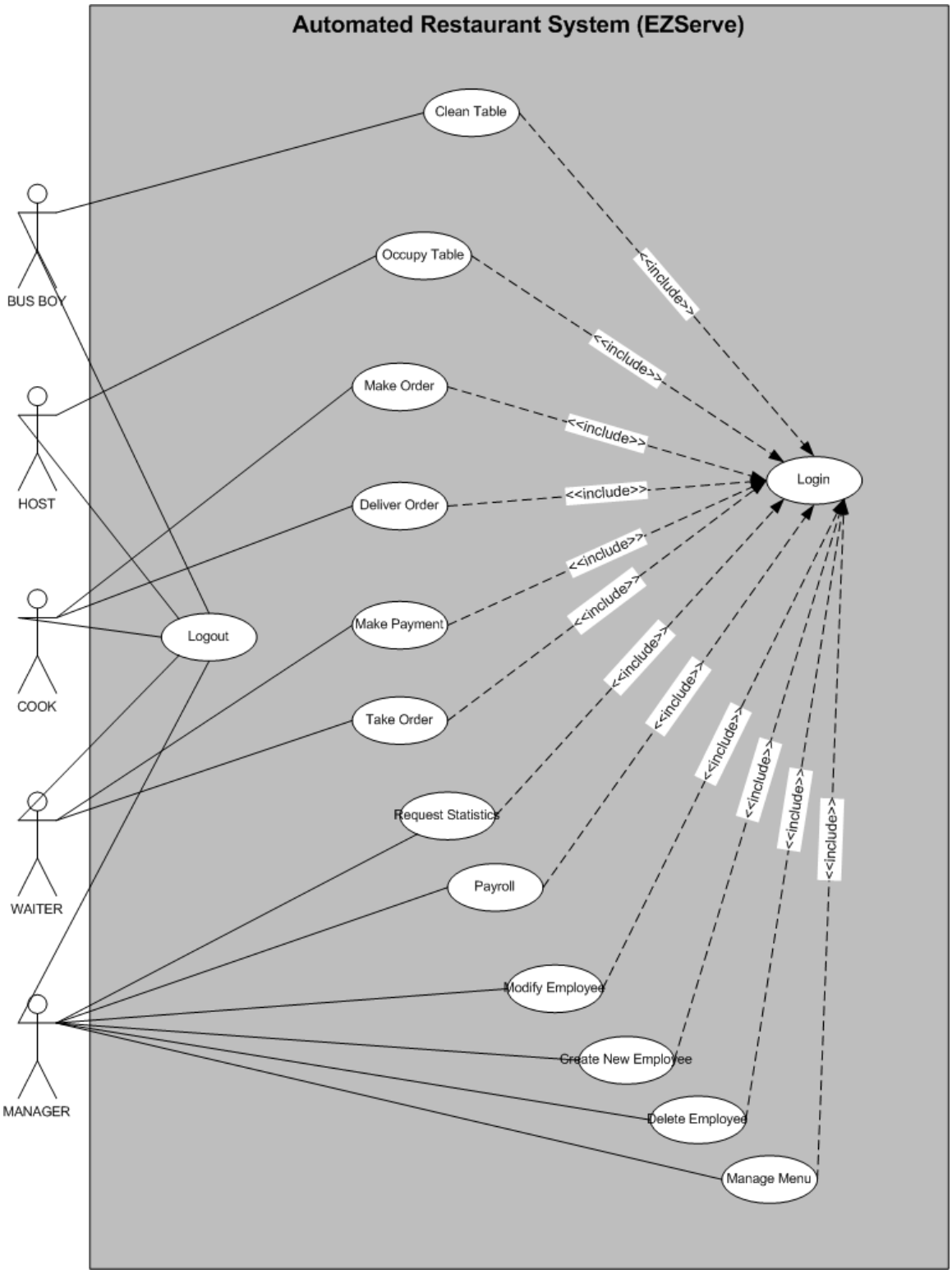


FURPS+

Usability	• The average employee will be more than qualified to operate the system. • The interface will be self explanatory. • Using standard login screen. • No documentation will be required to operate this system.
Reliability	• There will be security against users from trying to log into the system multiple time with the wrong password. • The system will be available to all the employees. • All employees will be given certain privileges to limit their access to certain data. • The system should not lose any of the data. The only exception being the data that the manager has manually deleted.
Performance	• The system will support only one user per terminal. • The host terminal and kitchen terminal will be updated periodically. • The waiter terminal will be updated at every log in. • Response time optimal limit is 100 milliseconds. Acceptable response time limit is 1000 milliseconds. • Online user limit is irrelevant to the restaurant industry, as it is in the magnitude of tens of thousands.
Supportability	• The system will be maintained by the Manager. • Possible Extension: Ability to update the Waiter of the status of an order without having to log in. • Possible Extension: Allowing the Manager to reorganize the table setting of the restaurant.
Implementation	• There is a financial constraint which requires a system that is financial feasible. • Server runs on UNIX, clients can run anything which can run Firebox and connect to a network (OS wise). • Hardware specifications are spelled out in great detail in the Hardware Requirements part of the 2nd report. In general a server and several terminal/client machines are necessary.
Interface	• The data will be imported by manager, host, waiter, busboy, and cook via their respective terminals. • The terminal is a touch screen. Resolution and other

	requirements are detailed in the Hardware Requirements part of the 2nd report. • GAOTek 2407 RFID PROXPOINT Card Reader must be equipped with each terminal.
Operation	• The manager of the restaurant will be managing the system.
Packaging	• The developers will install the system at the restaurant. • Possibly be simple enough for a restaurant to perform the installation alone.

Use Case Diagram



System Operation Contracts

Operation	Log In
Preconditions	Log in screen is displayed
Postconditions	Employee is logged in according with his/her privileges.

Operation	Occupy Table
Preconditions	tableStat = rdy
Postconditions	tableStat = busy

Operation	Make Order
Preconditions	Order available to be taken.
Postconditions	Order status changed to taken.

Operation	Clean Table
Preconditions	tableStat = dirty
Postconditions	tableStat = rdy

Operation	Deliver Order
Preconditions	Order is cooked.
Postconditions	orderStat = rdy

Operation	Close Tab
Preconditions	tableStat = busy
Postconditions	tableStat = dirty

Operation	Take Order
Preconditions	CUSTOMER request food item.
Postconditions	Order for food is created.

Operation	Request Statistics
Preconditions	None
Postconditions	Requested Statistics are displayed.

Operation	Payroll
Preconditions	MANAGER logs in.
Postconditions	PAYROLL is modified or not.

Operation	Modify Employee
Preconditions	MANAGER requests employee modification
Postconditions	Employee data is modified

Operation	Create New Employee
Preconditions	MANAGER requests employee addition
Postconditions	Employee is added to the staff list and payroll

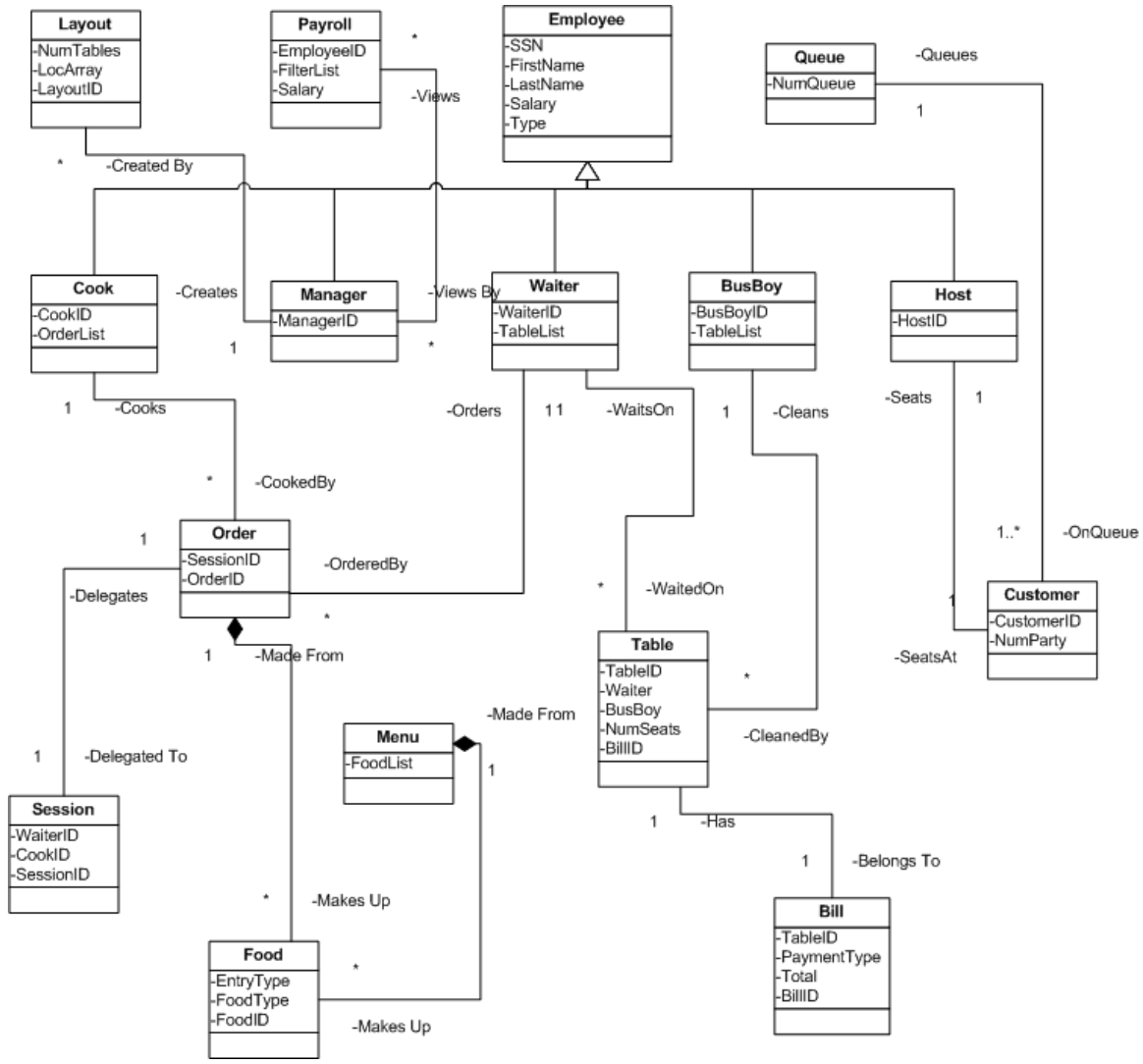
Operation	Delete Employee
Preconditions	MANAGER requests employee removal
Postconditions	Employee is removed from payroll and staff list

Operation	Logout
Preconditions	EMPLOYEE is logged in
Postconditions	EMPLOYEE is logged out

Operation	Add Menu Item
Preconditions	Item Entry and Food Category exists
Postconditions	Menu Item created

Operation	Delete Menu Item
Preconditions	Item exists on the menu
Postconditions	Item is deleted from the menu

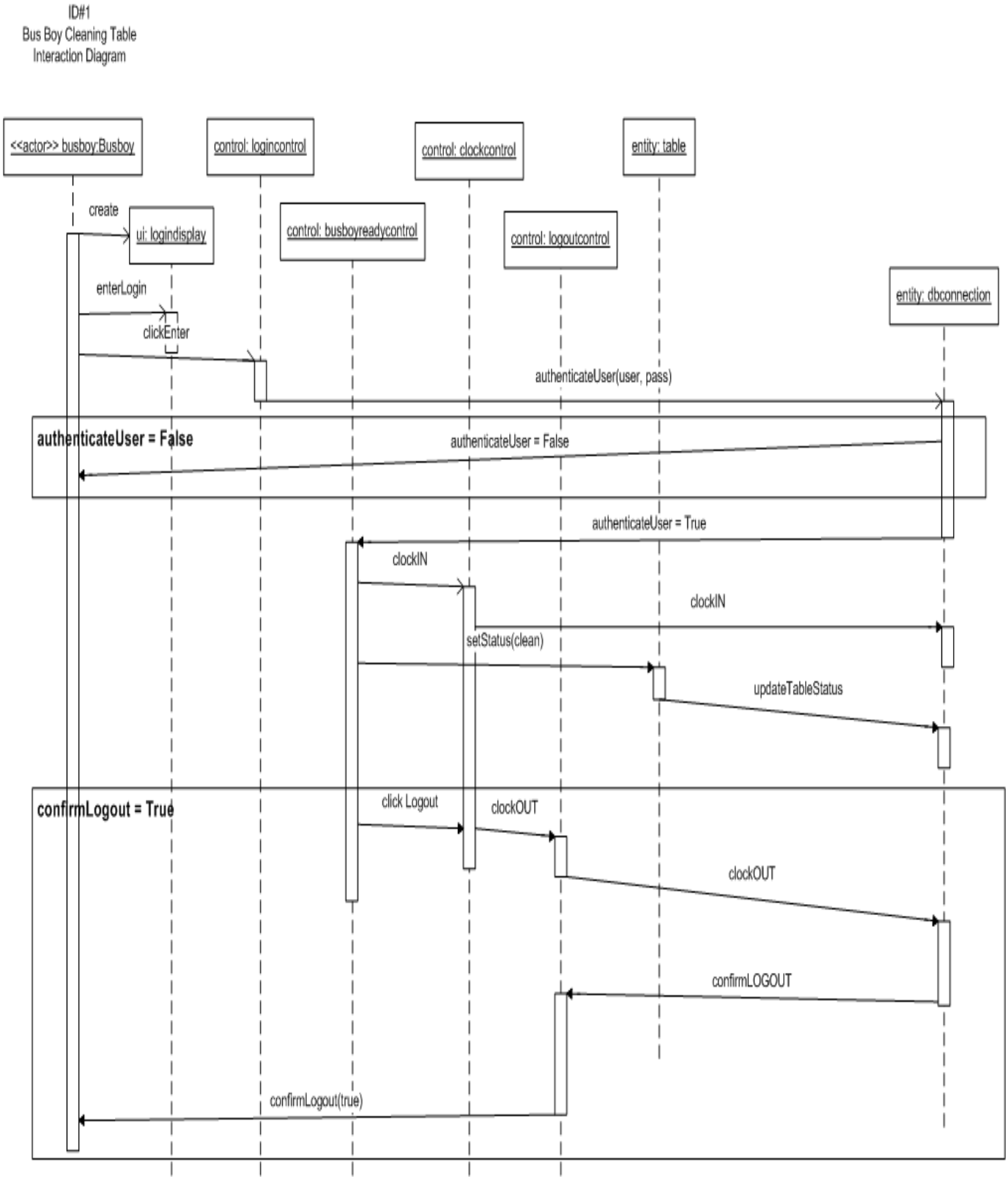
Domain Model



Interaction Diagrams

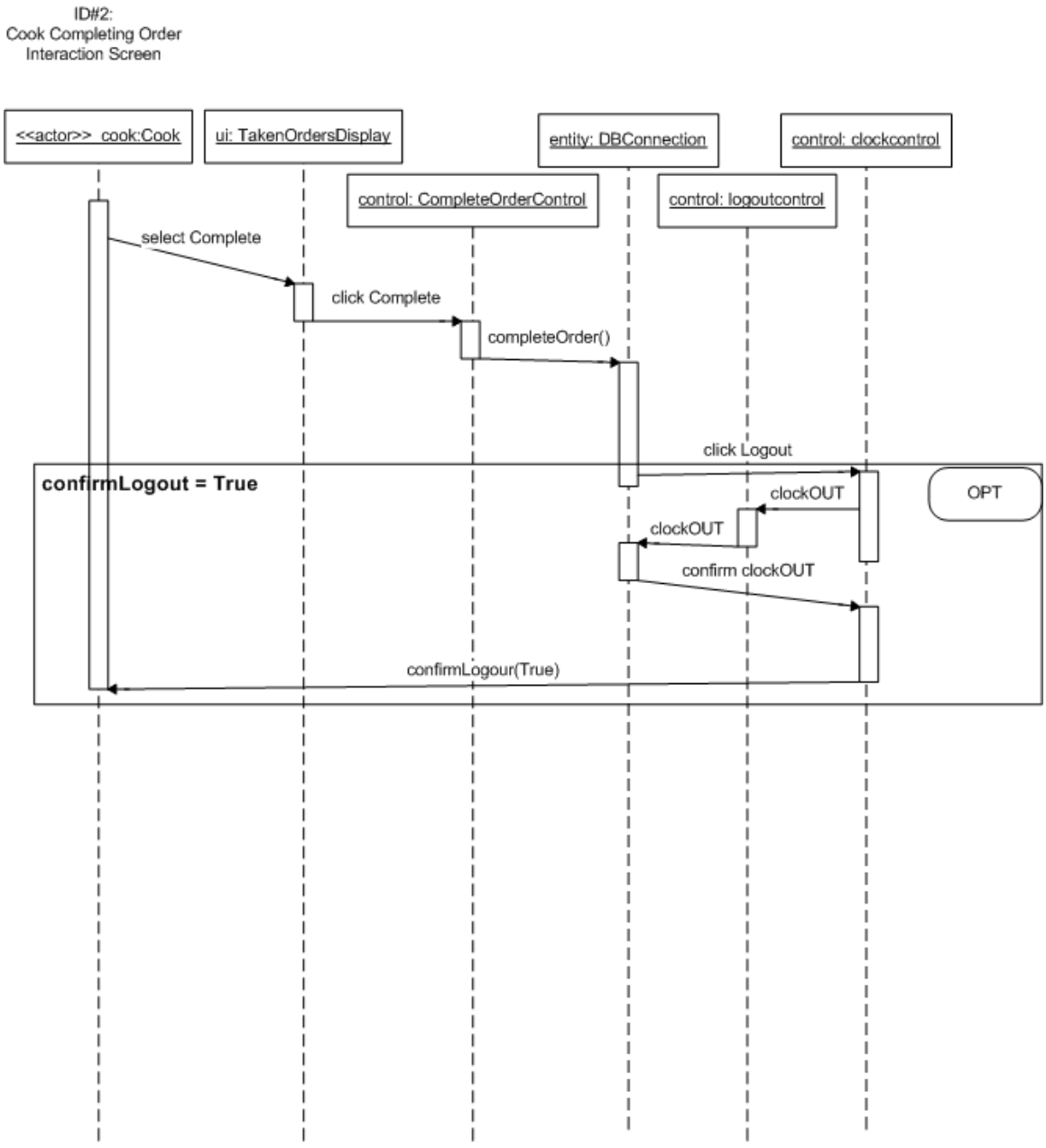
We are using the model-view-control architecture. Hence, everything is separated into view, control, and model objects. The view objects render all the objects on the screen, including the presentations of the control objects such as buttons. In accordance with this principle, the actor interacts with the system through clicking the control objects. The control objects than, either opens a new view object or contacts a model (entity object) to perform the needed calculations. This ensures the principles of expert doer (that who knows should do the task), high cohesion (do not take on too many responsibilities of Type 2(computation)), and low coupling (do not take on too many responsibilities of Type 3(communication)).

ID #1 Clean Table by Busboy



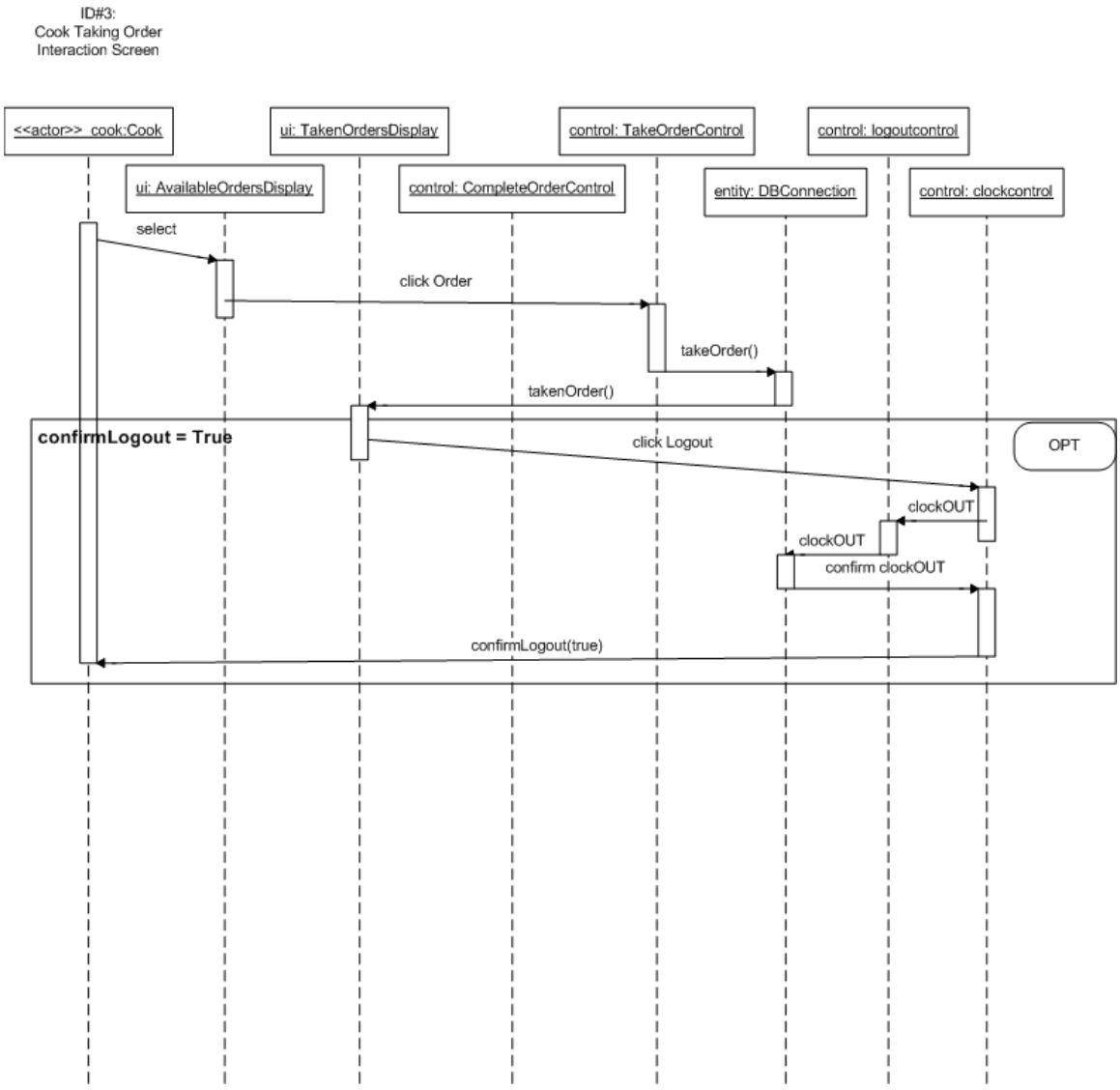
This model-view-control architecture can be seen in Interaction Diagram #1 (Bus Boy Cleaning Table Interaction Diagram). The actor, bus boy, interacts with the system by either clicking the enter button to login or by selecting a table on the user interface, and clicking ready. This than creates the event that the control objects will proceed with the necessary steps to complete the desired task. For instance, during the login phase the login control object will authenticate the user's name and password by accessing the entity, dbconnection. From this interaction the user's information will either be given a true value or false value depending on the entered information.

ID #2 Complete Order by Cook



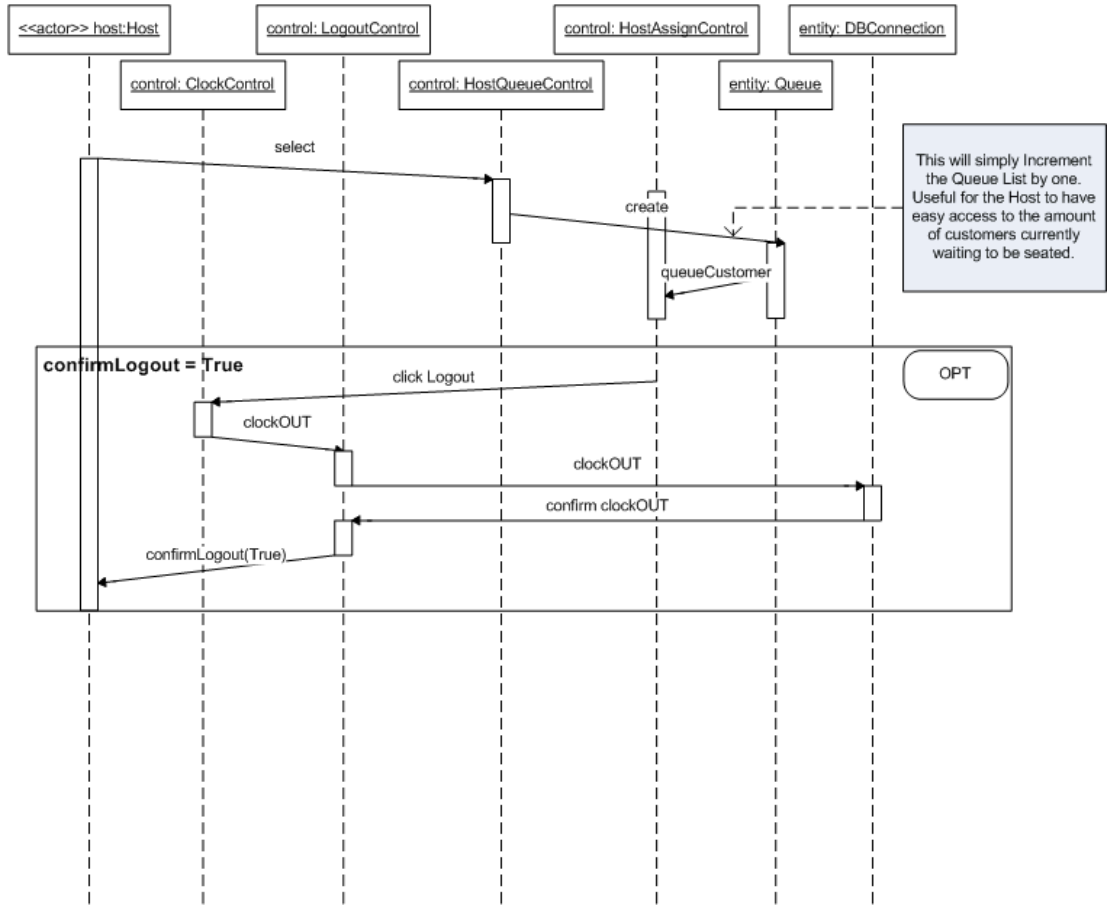
The model-view-control architecture can be seen in Interaction Diagram #2 (Cook Completing Order Interaction Diagram). The actor, cook, interacts with the system by either clicking the enter button to login or by selecting the appropriate order and than clicking complete button. This than creates the event that the control objects will proceed with the necessary steps to complete the desired task. For instance, during the “complete table” phase the actor, cook, will interact with the control by selecting the order and clicking complete. This will create an event for the control TakeOrderControl to access the entity, Order, and than access the DBConnection to update the database with the appropriate information.

ID #3 Take Order by Cook

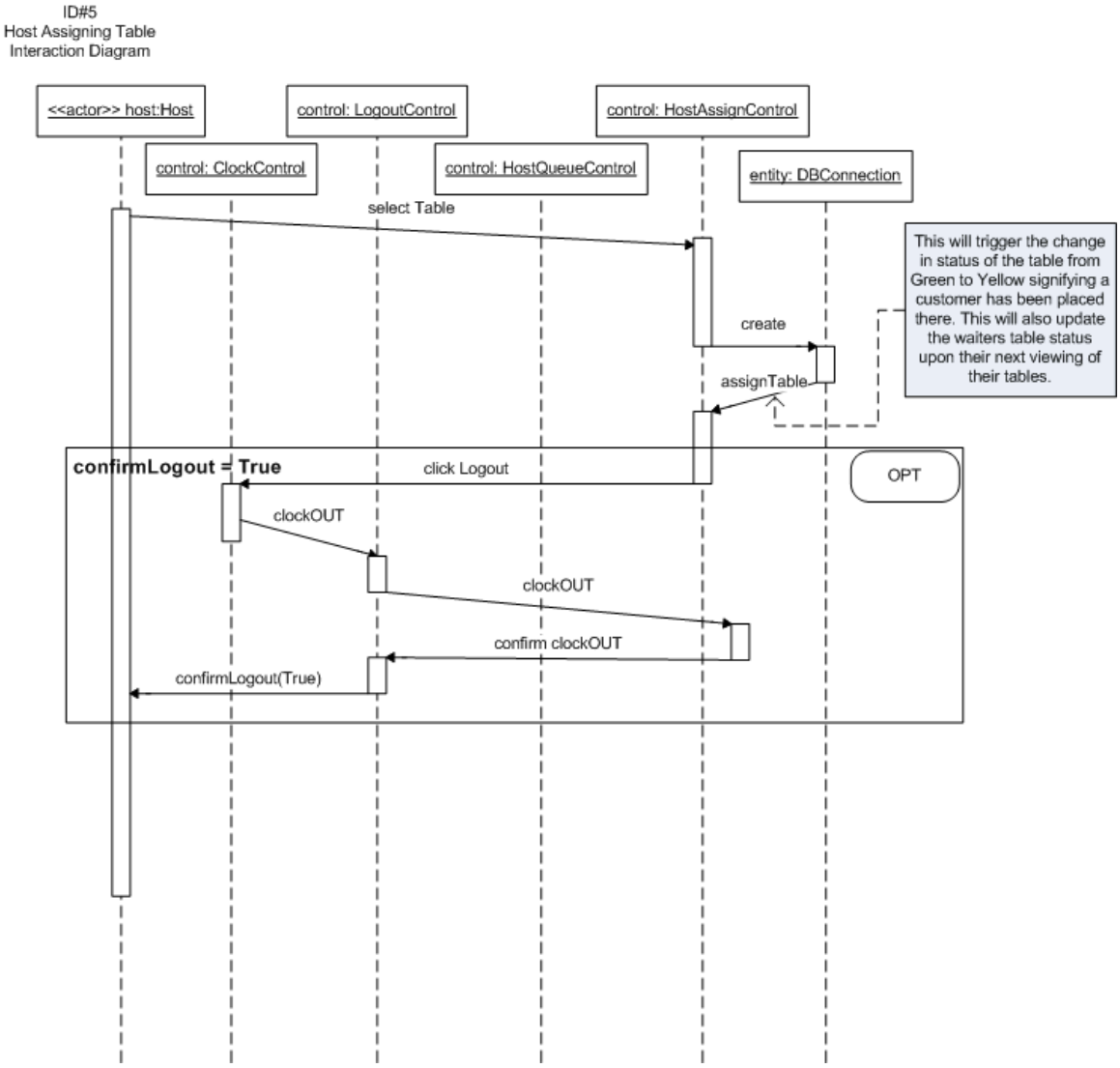


ID #4 Queue Customers by Host

ID#4
Host Queueing
Interaction Diagram
List

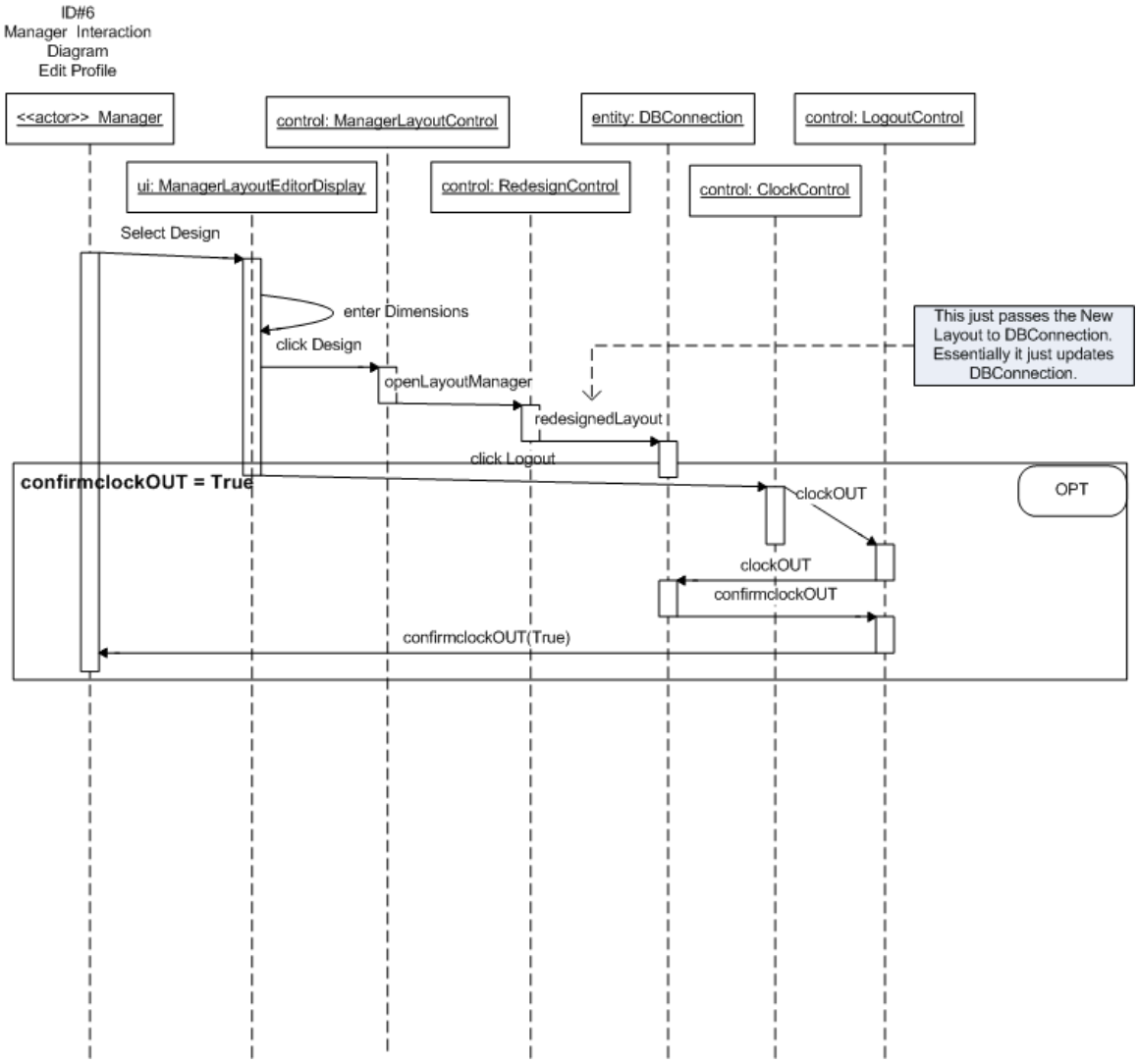


ID #5 Customer Assignments to Table by Host

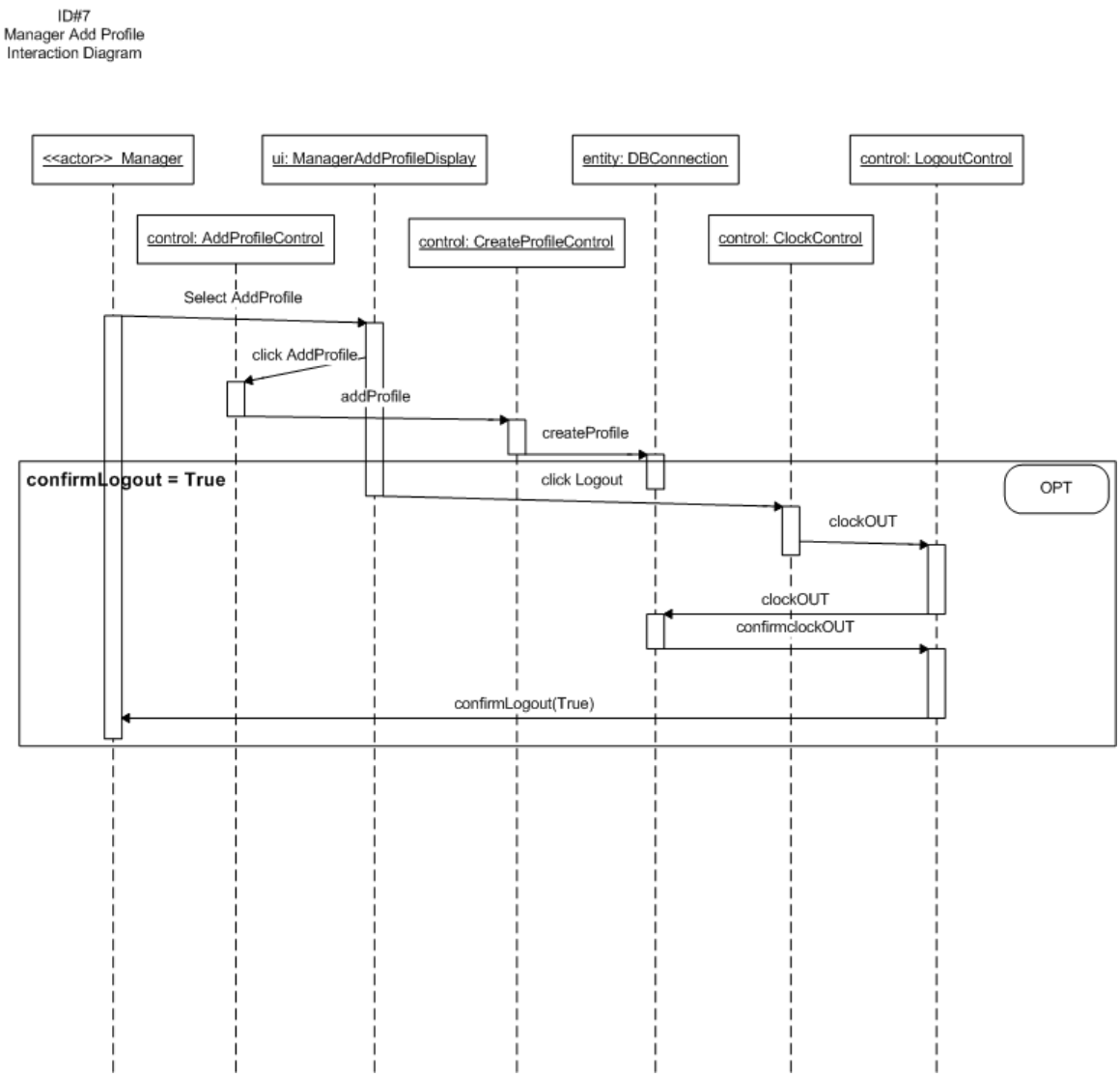


The model-view-control architecture can be seen in the Interaction Diagram #5 (Host Assigning Table Interaction Diagram). The actor, host, interacts with the system by either clicking the enter button to login or by selecting the desired table and clicking assigning button. This than creates the event that the control objects will proceed with the necessary steps to complete the desired task. For instance, during the “assign table” phase the actor, host, will interact with the control by selecting the desired table and clicking assign. This will create an even for the control HostAssignControl to access the entity, Table, and than access the DBConnection to update the database with the appropriate information.

ID #6 Layout Design by Manager



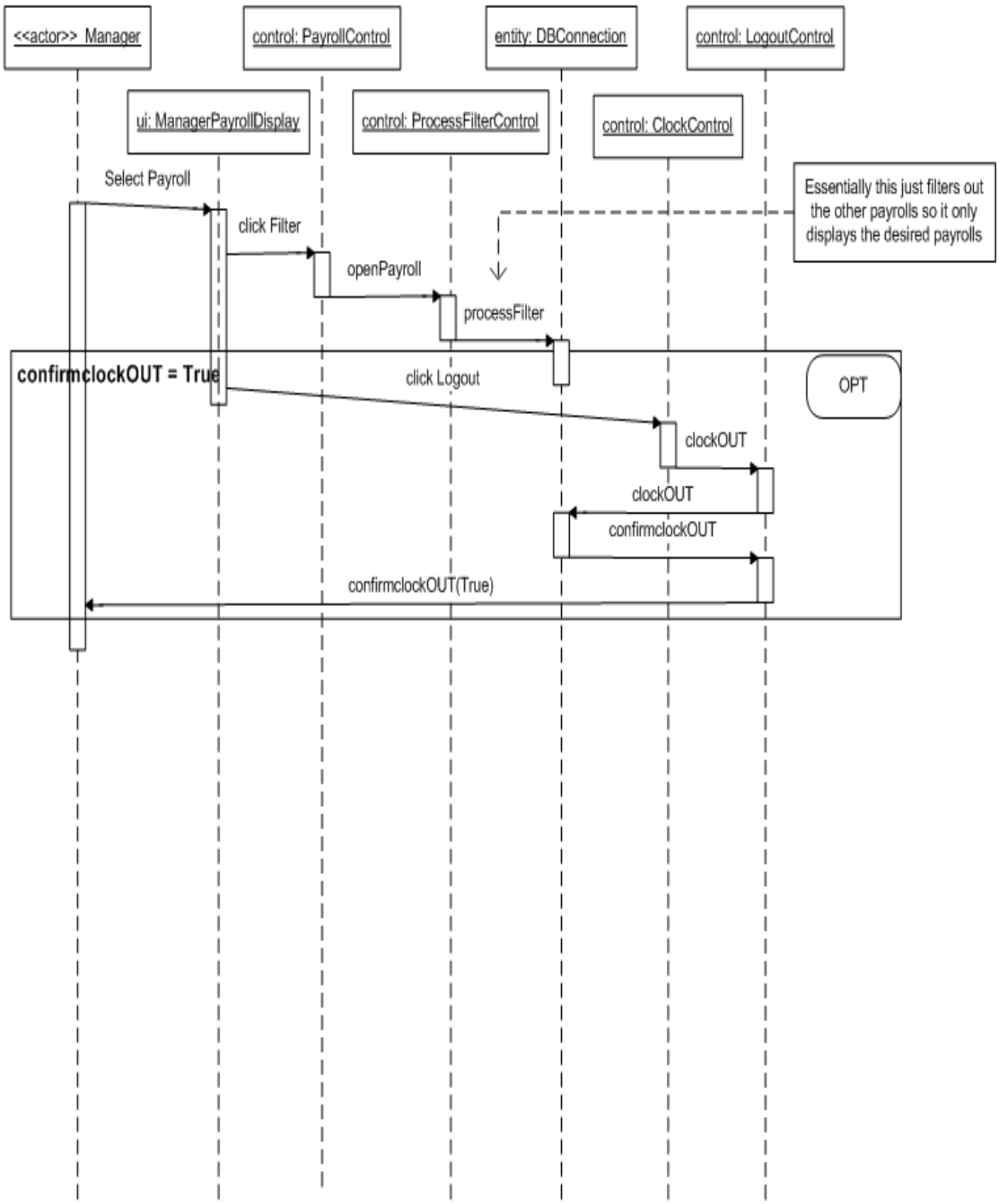
ID #7 Add Profile by Manager



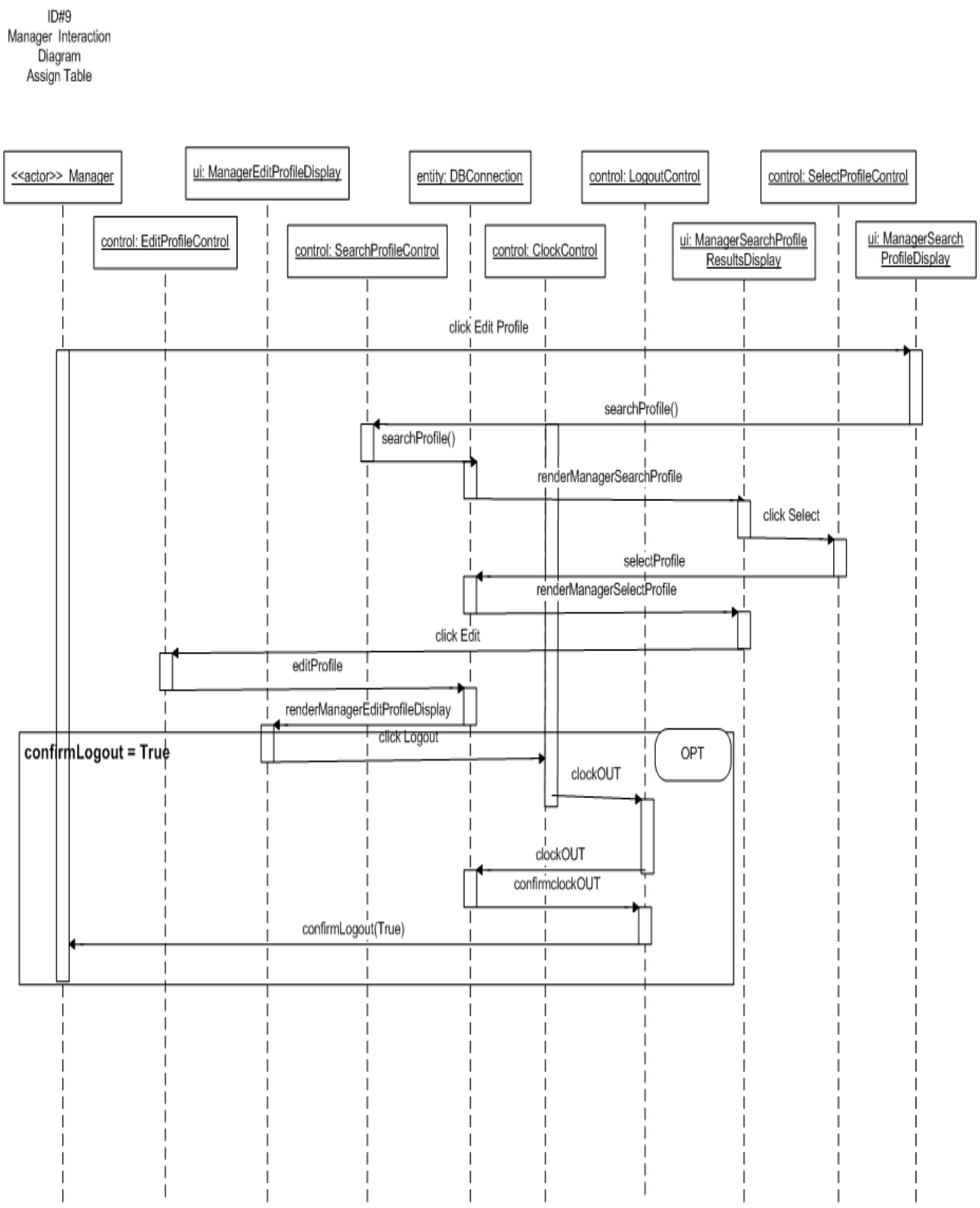
The model-view-control architecture can be seen in the Interaction Diagram #7 (Manager Add Profile Interaction Diagram). The actor, manager, interacts with the system by either clicking the enter button to login or by selecting the AddProfile button. This than creates the event that will spawn a new screen with a blank profile, which will be filled out by the actor, and than click the add button. This will than access the Dbconnection which will be updated appropriately.

ID #8 View Payroll by Manager

ID#8
Manager Interaction
Diagram
Payroll

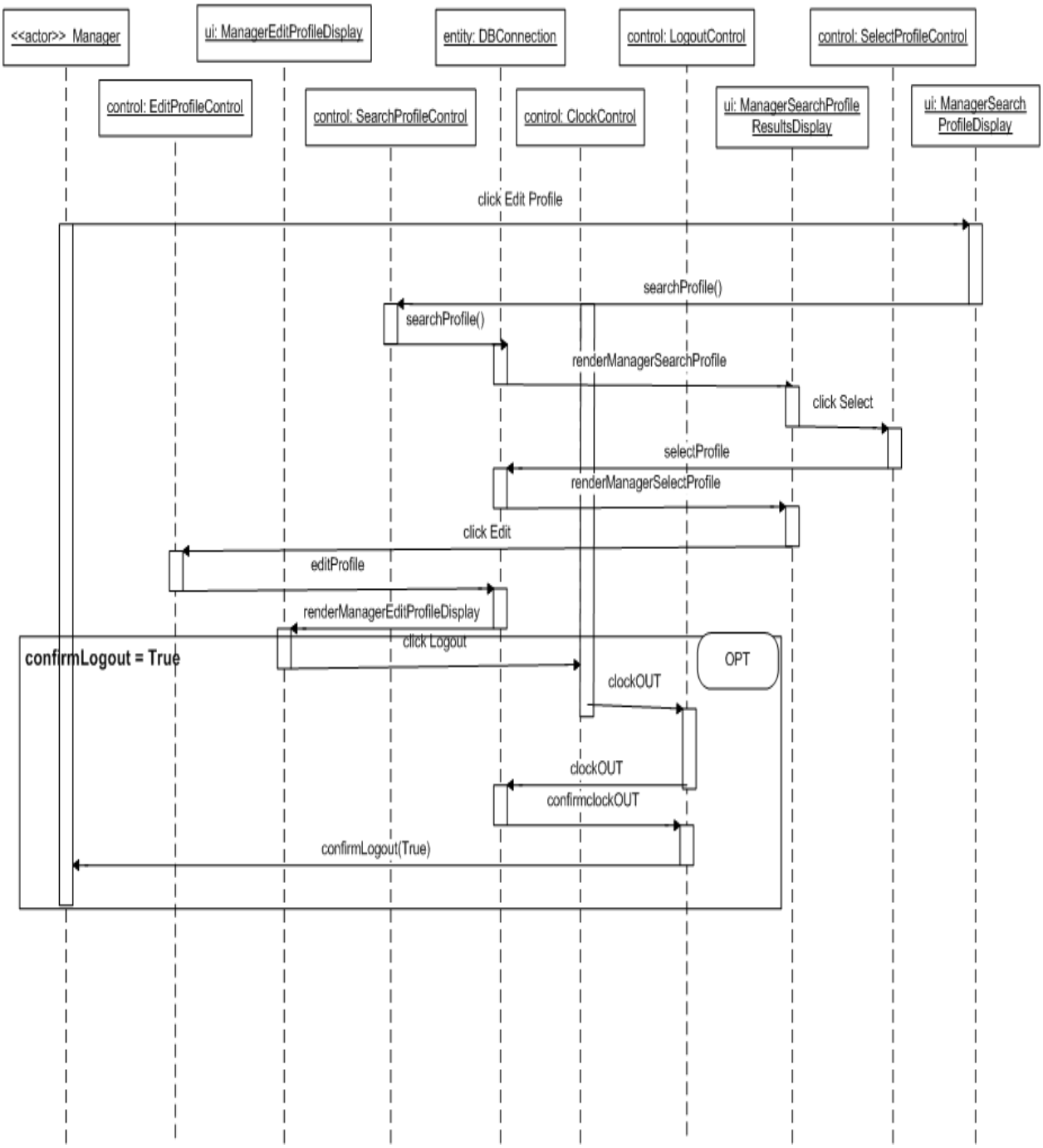


ID #9 Assign Table by Manager



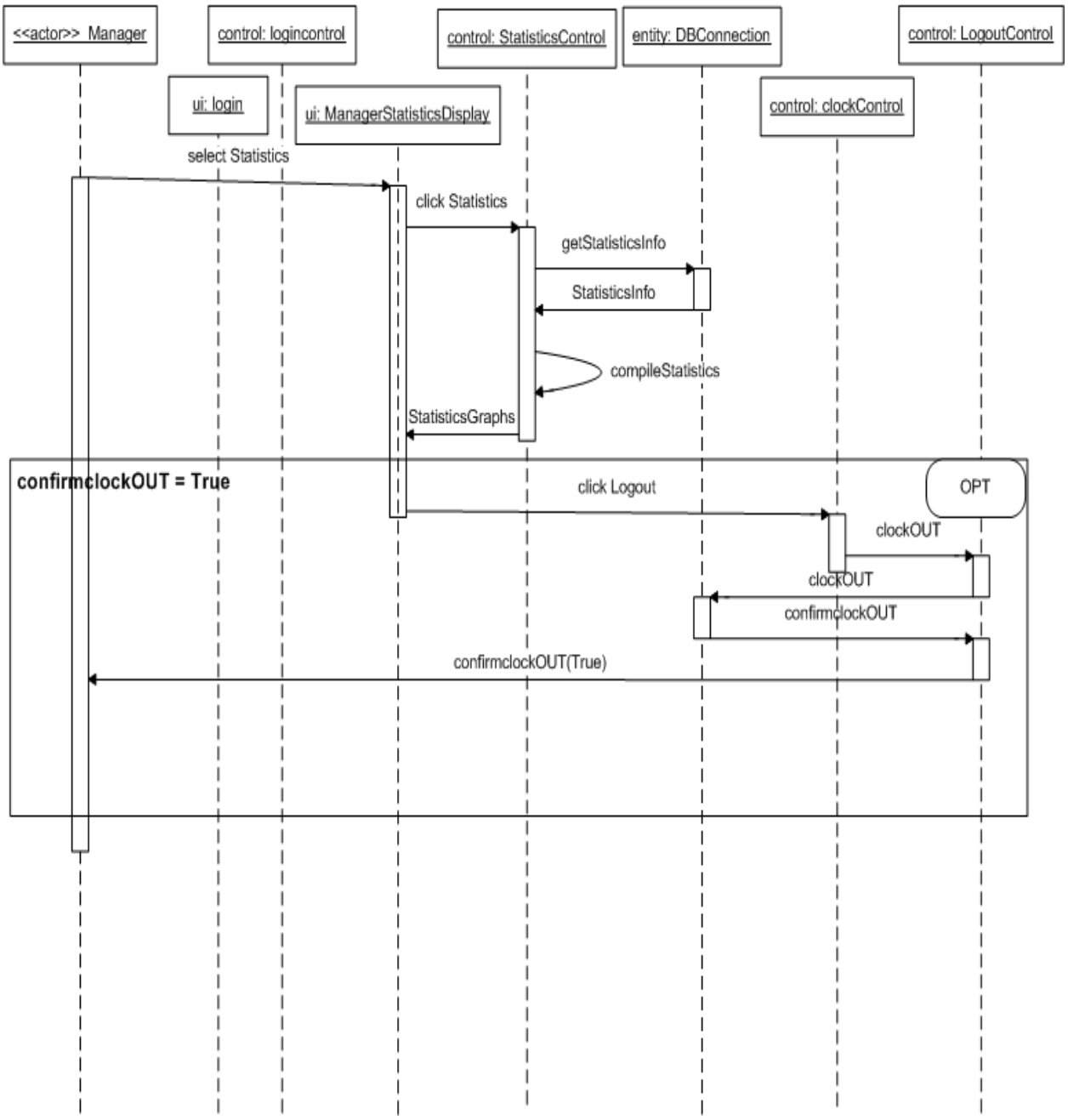
ID #10 Edit Profile by Manager

ID#9
Manager Interaction
Diagram
Edit Profile

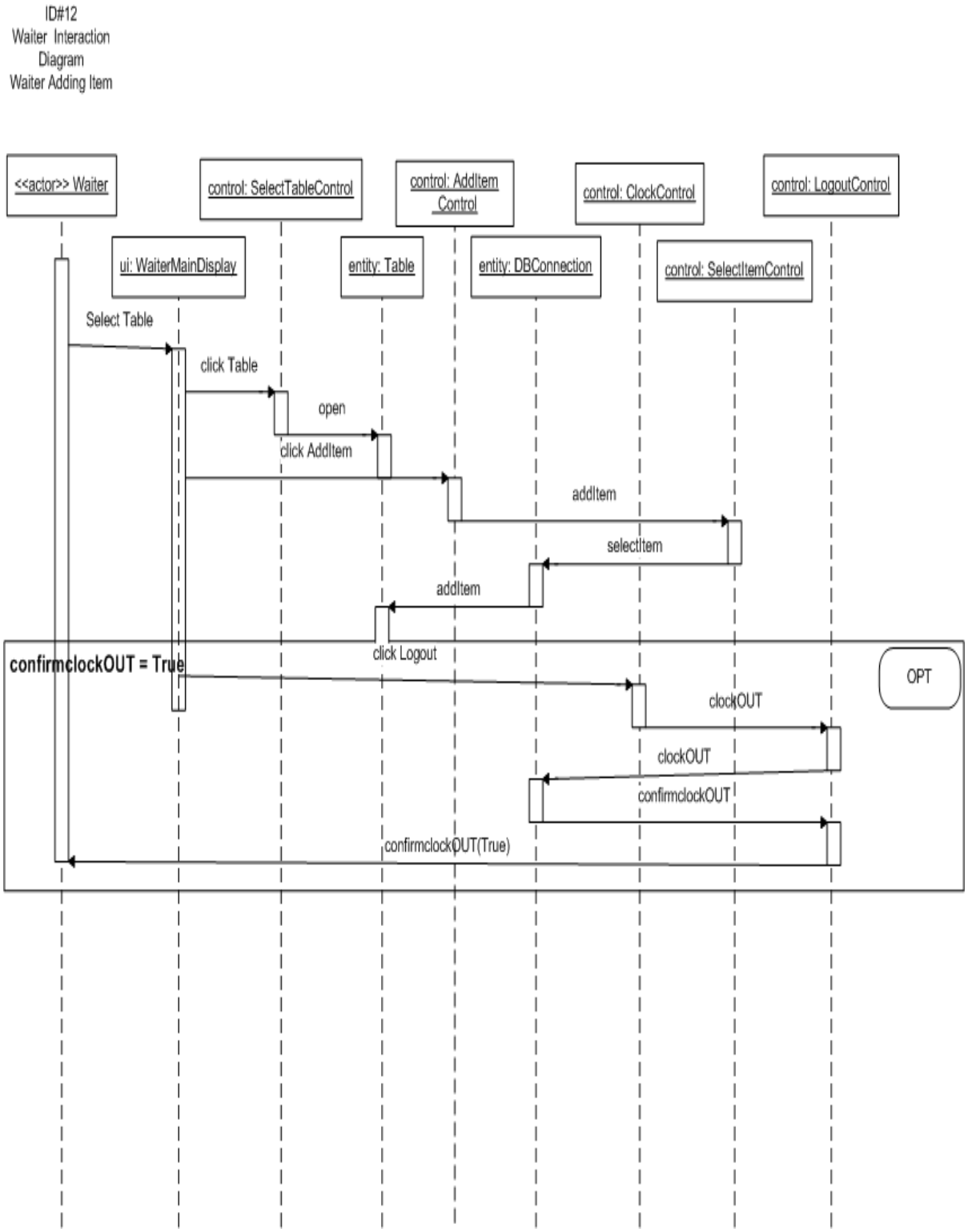


ID #11 View Statistics by Manager

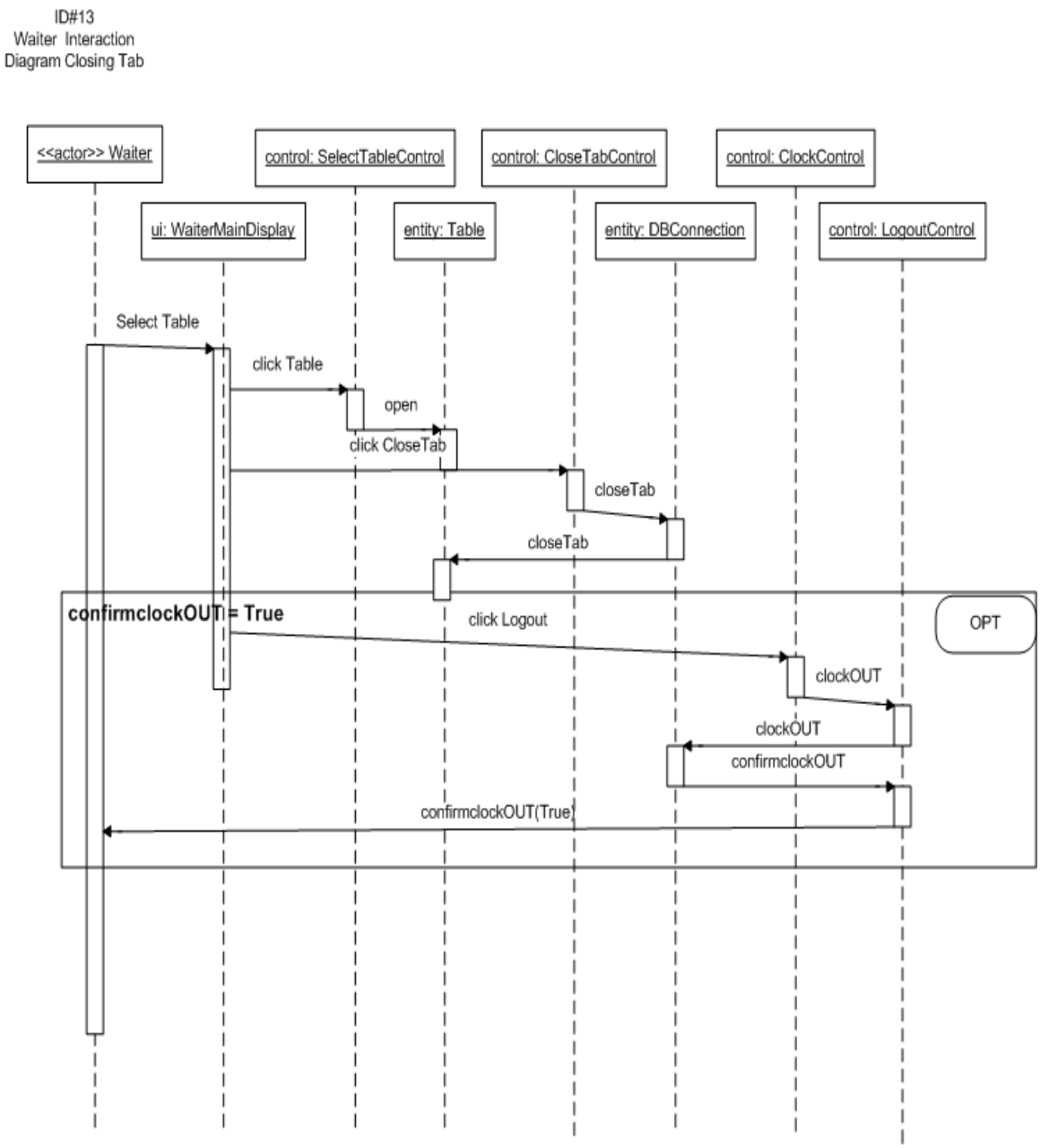
ID#11
Manager Interaction
Diagram
Statistics



ID #12 Add Item by Waiter

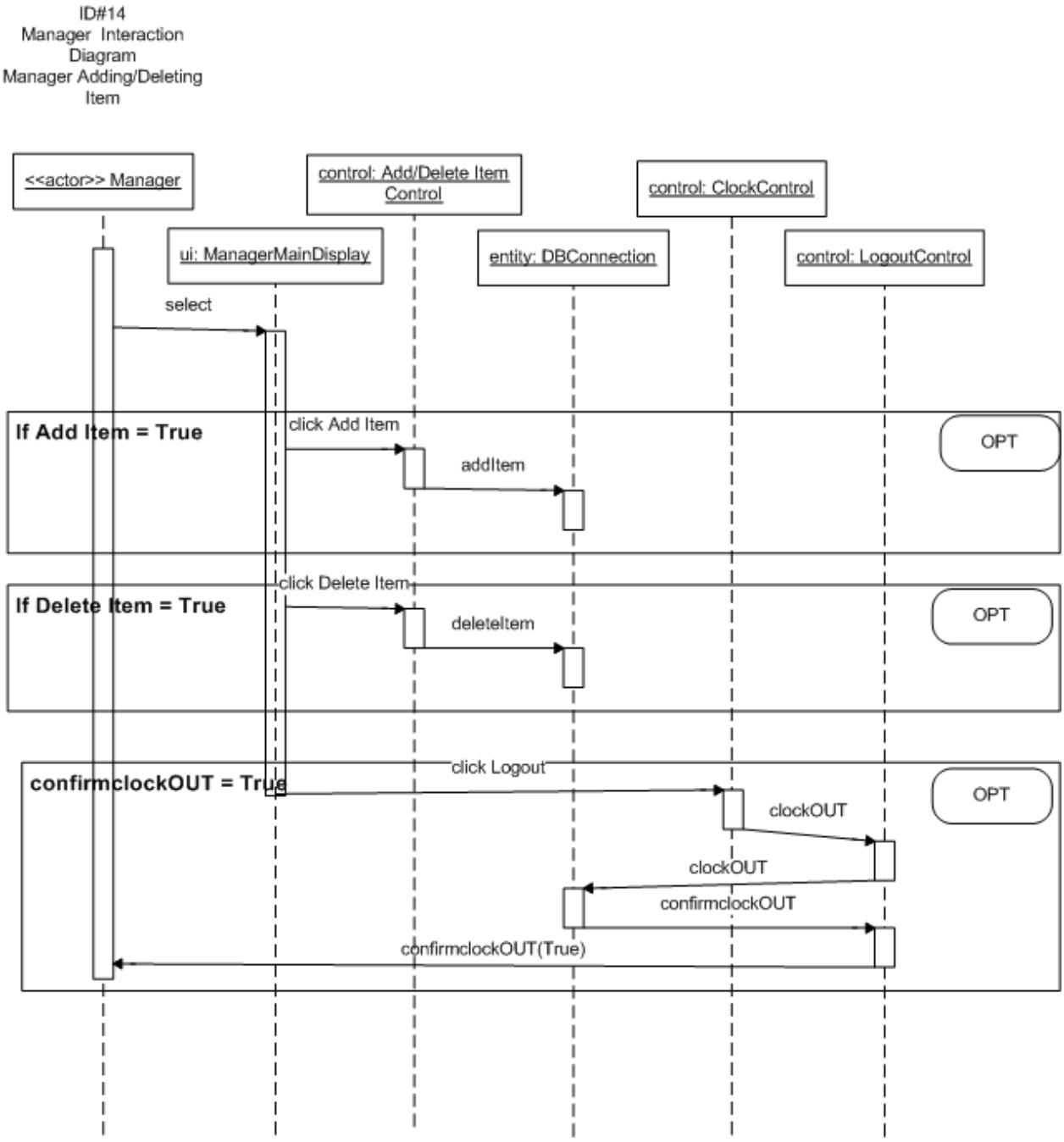


ID #13 Close Tab by Waiter



The model-view-control architecture can be seen in the Interaction Diagram #13 (Waiter Closing Tab Interaction Diagram). The actor, waiter, interacts with the system by either clicking the enter button to login or by selecting the CloseTab button. This will than create an even that will access the CloseTabControl which will than access the DBConnection and to update the database with the appropriate information.

ID #14 Manage Menu By Manager



The model-view-control architecture can be seen in the Interaction Diagram #14 (Manager Adding/Deleting Menu Item Interaction Diagram). The actor, Manager, interacts with the system by either clicking the enter button to login or by selecting the Add Menu button or Delete Menu button. This will than create an event that will access the AddMenuControl or DeleteMenuControl which will than access the DBConnection and to update the database with the appropriate information.

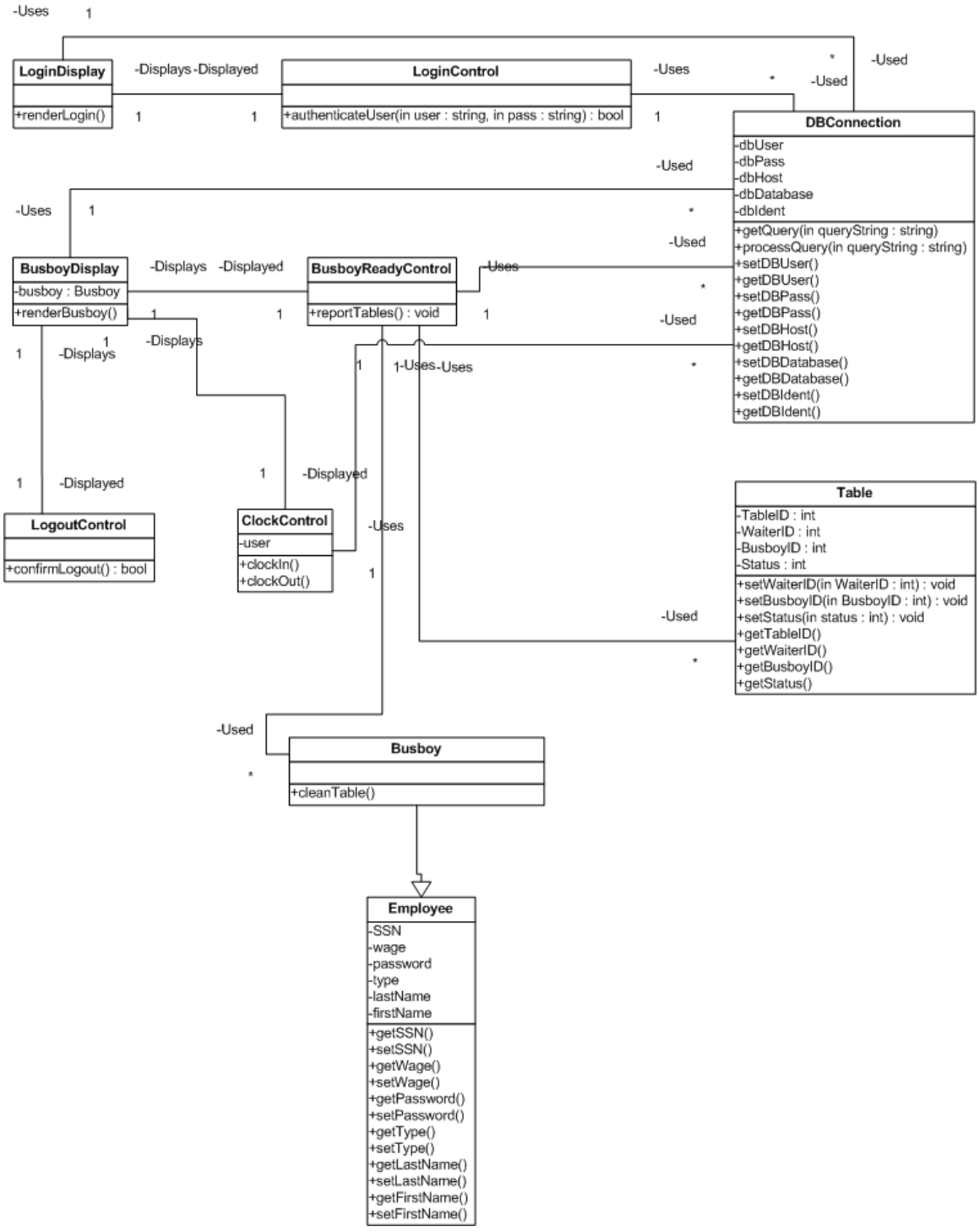
Class Diagrams and Interface Specifications

Class Diagrams

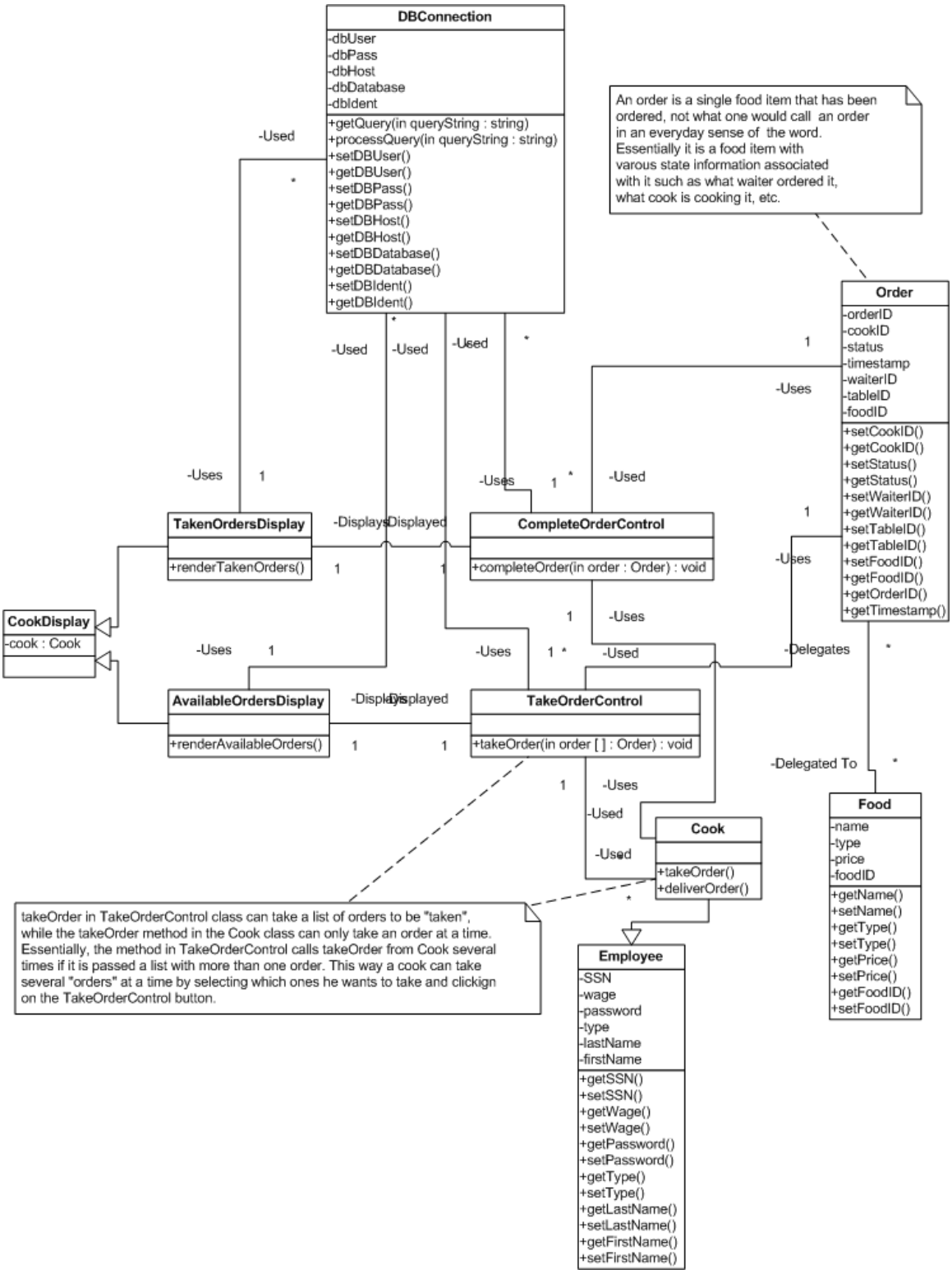
Because of the clutter that resulted in an attempt to keep the class diagram to one page, we have broken up the class diagram into several separate pages, with some of the classed appearing multiple times on different pages for illustration purposes. The division naturally occurred along the user package lines. It will be easily evident upon inspection.

The subdivided class diagrams appear on the following pages, one per page.

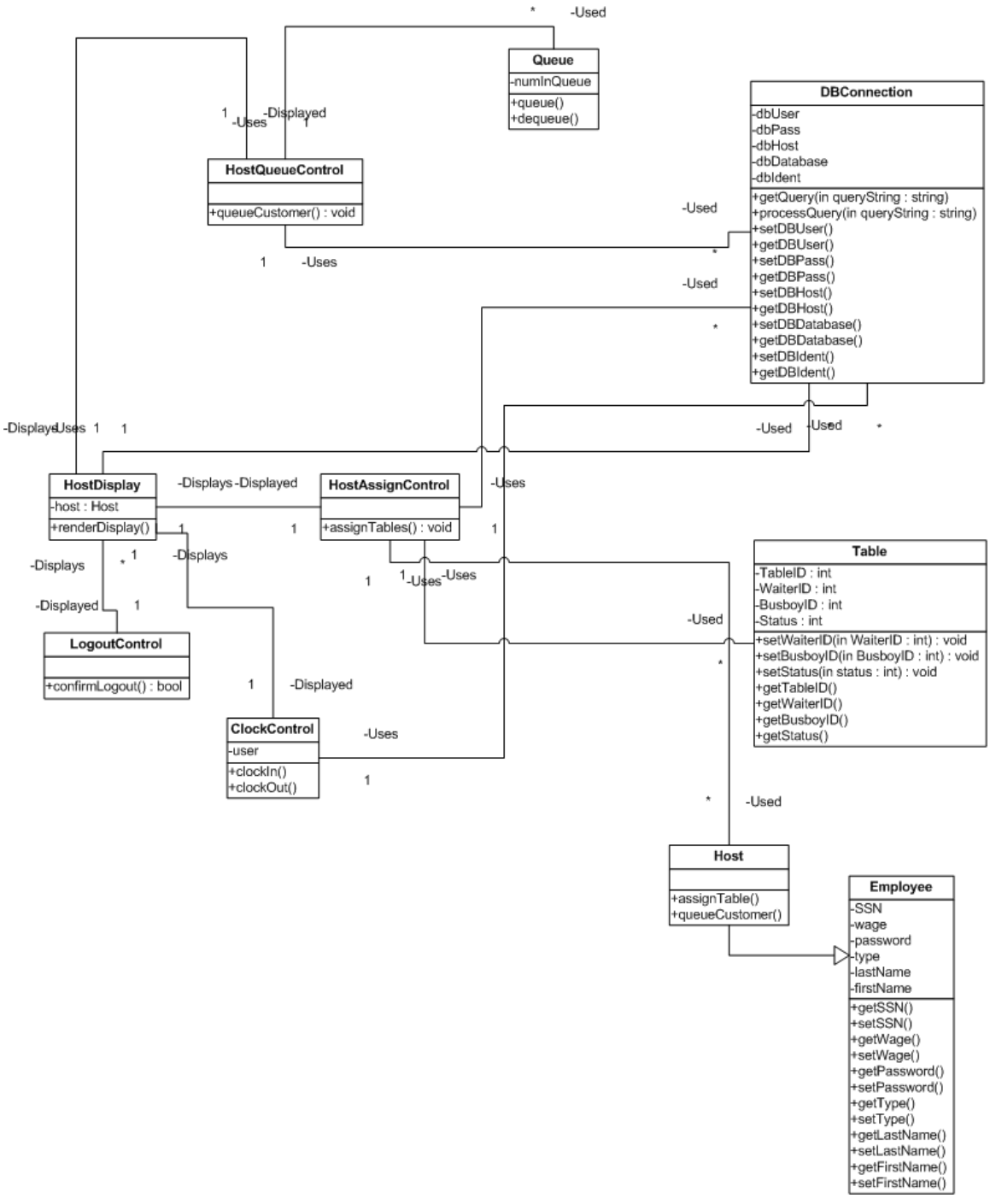
Login and Busboy Class Diagram



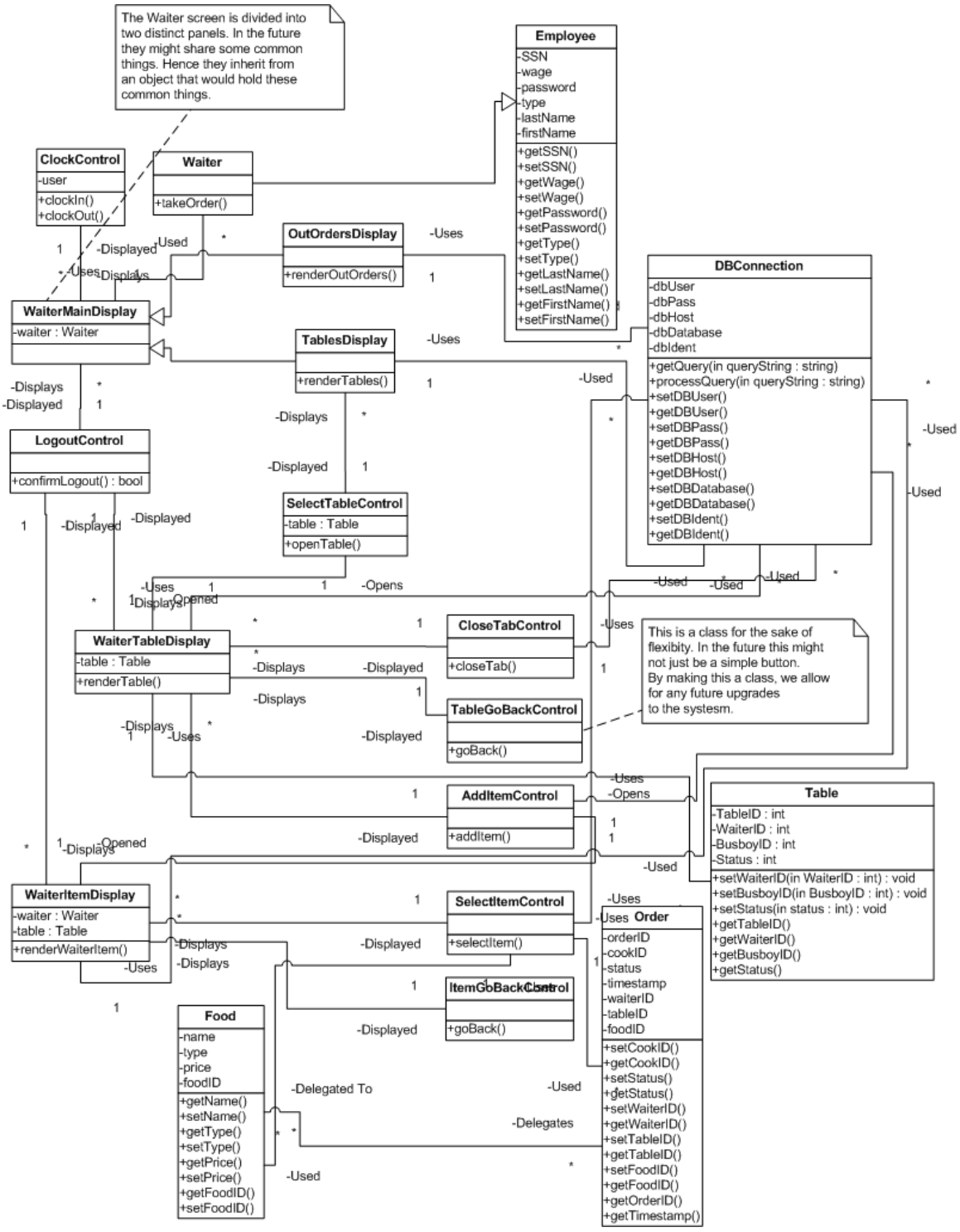
Cook Class Diagram



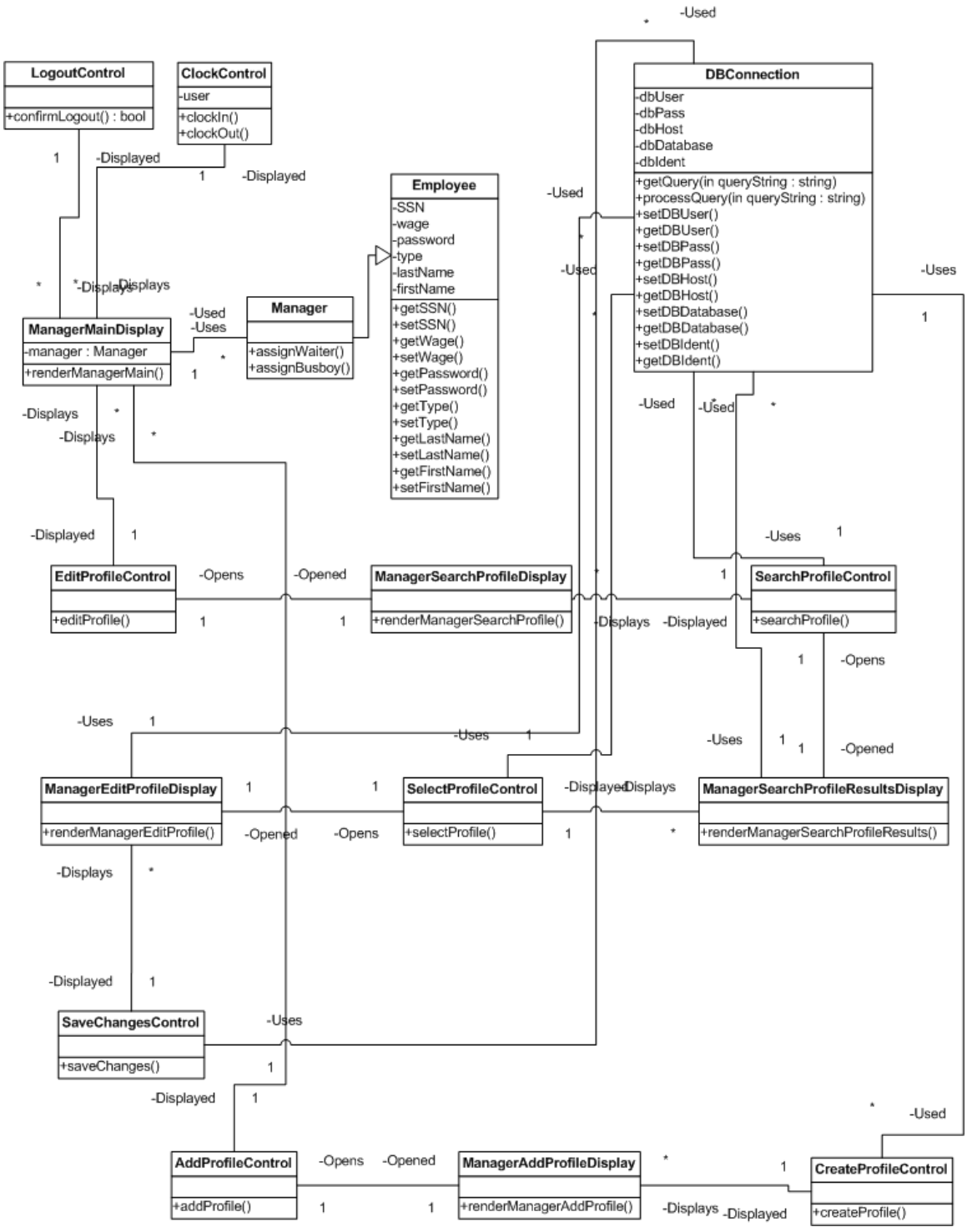
Host Class Diagram



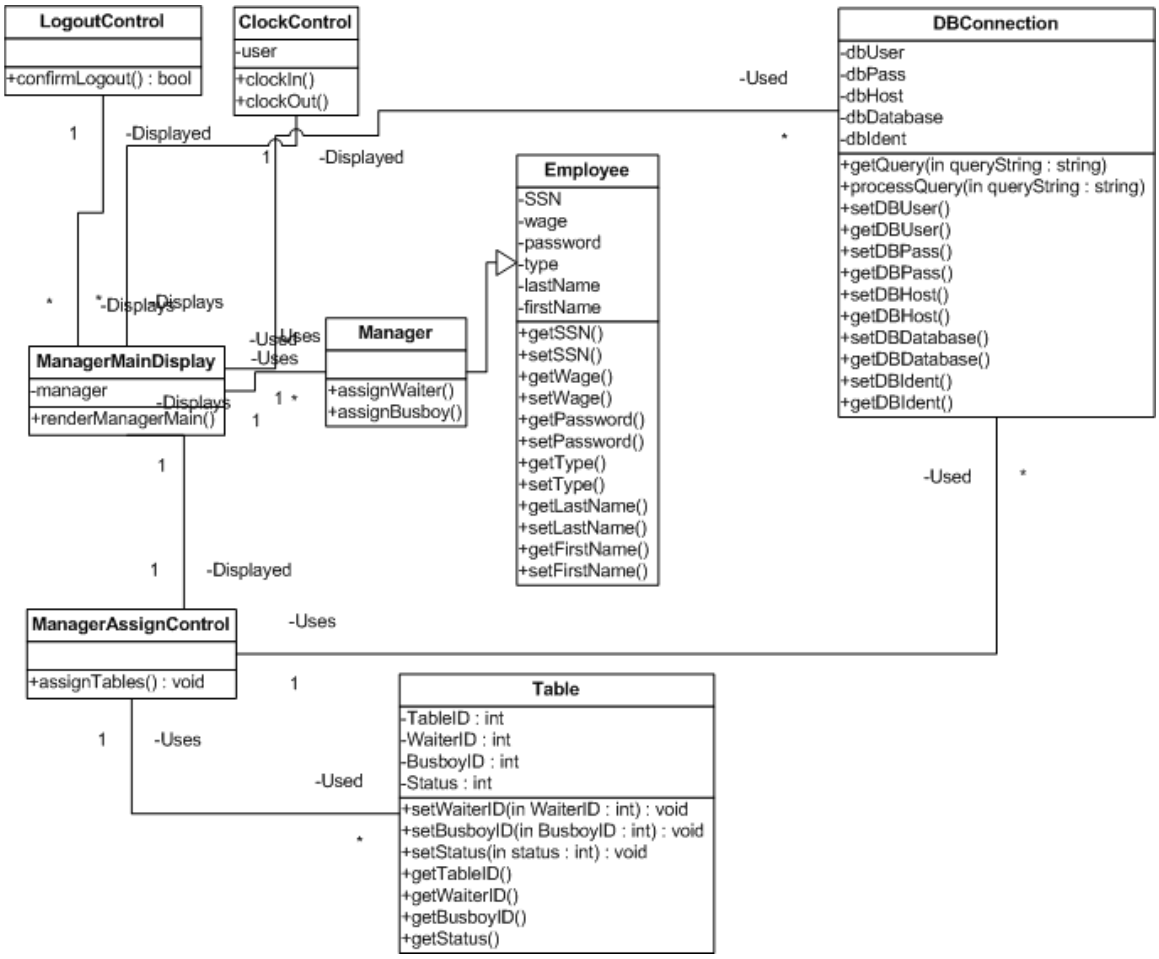
Waiter Class Diagram



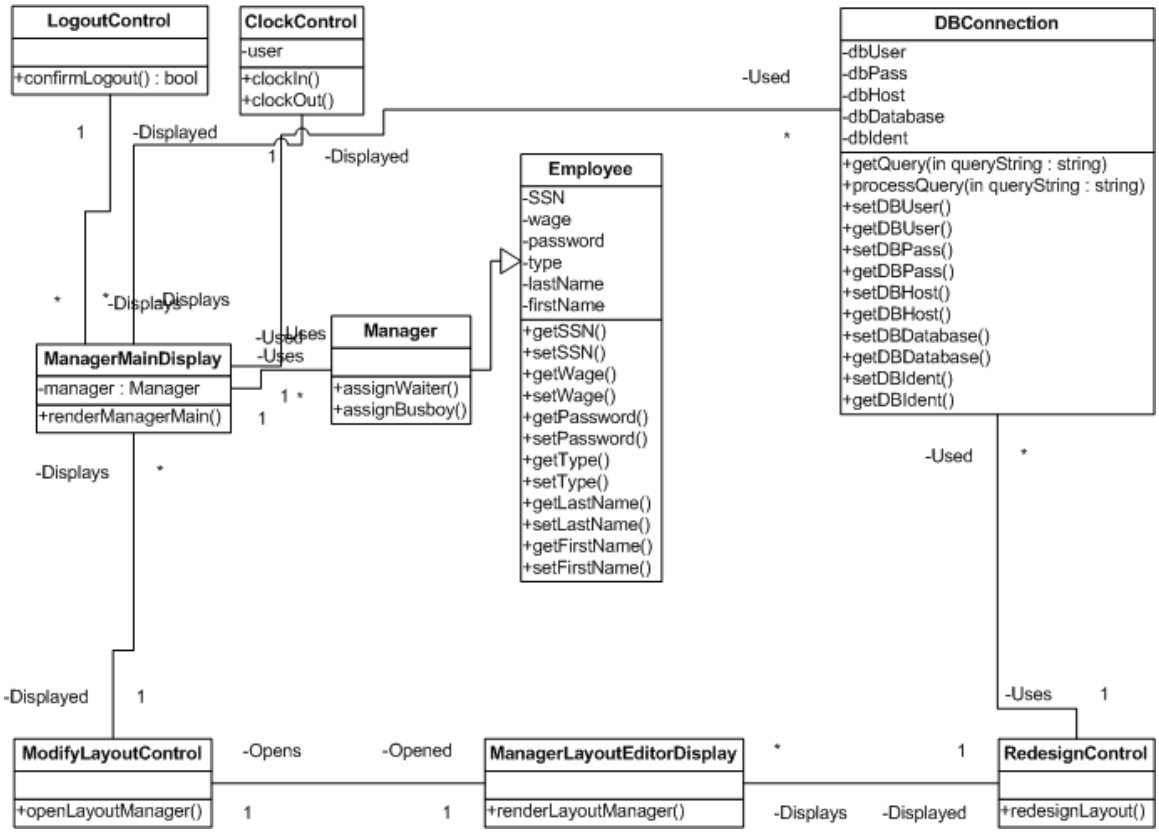
Manager Add/Edit Profile Class Diagram



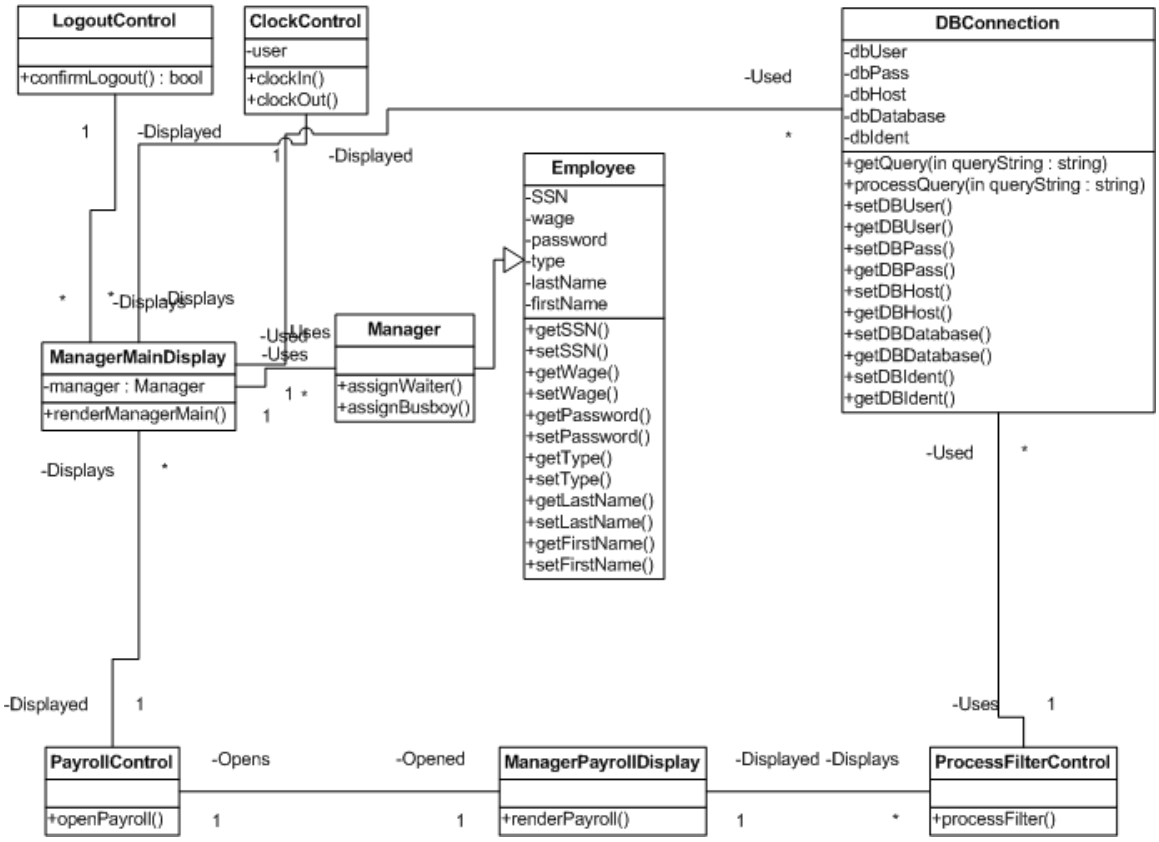
Manager Assign Tables Class Diagram



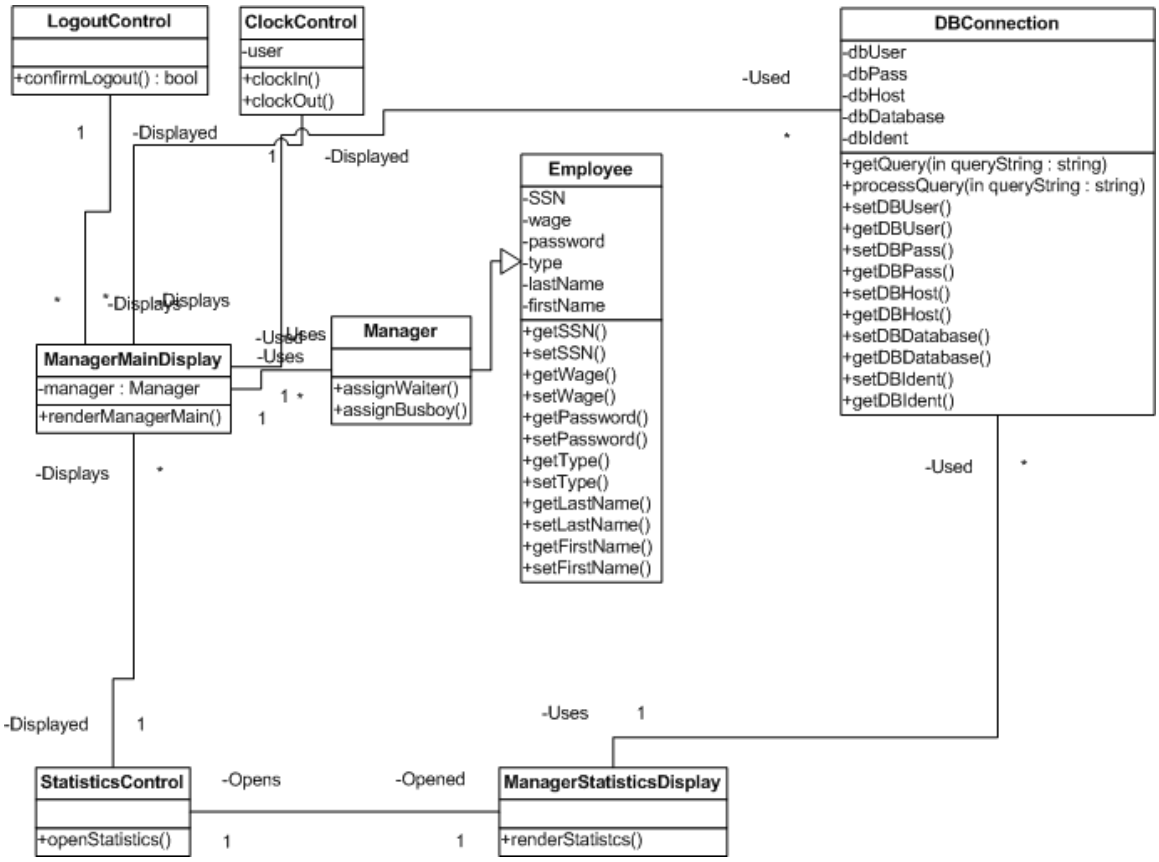
Manager Layout Editor Class Diagram



Manager Payroll Class Diagram



Manager Statistics Class Diagram



Data Types and Operation Signatures

Package: Utility

LoginDisplay

Attributes: N/A

Operations:

- + renderLogin() : void

LoginControl

Attributes: N/A

Operations:

- + authenticateUser(user: string, pass: string) : boolean

LogoutControl

Attributes: N/A

Operations:

- + confirmLogout() : boolean

ClockControl

Attributes:

- - user : Employee

Operations:

- + clockIn() : void
- + clcokOut() : void

Table

Attributes:

- - TableID : int
- - BusboyID : int
- - WaiterID : int
- - Status : int

Operations:

- + getTableID() : int
- + setWaiterID(waiterID: int) : void
- + getWaiterID() : int
- + setBusboyID(busboyID : int) : void
- + getBusbodyID() : int
- + setStatus(status : int) : void
- + getStatus() : int

Order

Attributes:

- - orderID : int
- - cookID : int
- - status: int
- - timestamp : MySQL timestamp
- - waiterID : int
- - tableID: int
- - foodID: int

Operations:

- + setCookID(cookID : int) : void
- + getCookID() : int
- + setStatus(status : int) : void
- + getStatus() : int
- + setWaiterID(waiterID : int) : void
- + getWaiterID() : int
- + setTableID(tableID : int) : void
- + getTableID() : int
- + setFoodID(foodID: int) : void
- + getFoodID() : int
- + getTimestamp() : MySQL timestamp
- + getOrderID() : int

Food

Attributes:

- - name : string
- - type : int
- - price : double
- - foodID : int

Operations:

- + getType() : int
- + setType(type : int) : void
- + getName() : string
- + setName(name : string) : void
- + getPrice() : double
- + setPrice(price : double) : void
- + getFoodID() : int
- + setFoodID(foodID : int) void

Package: Database

DBConnection

Attributes:

- - dbUser : string
- - dbPass : string
- - dbHost : string
- - dbDatabase : string
- - dbIdent : string

Operations:

- + getQuery(queryString : string) : Associative Array
- + processQuery(queryString : string) : void
- + setDBUser(user : string) : void
- + getDBUser() : string
- + setDBPass(pass : string) : void
- + getDBPass() : string
- + setDBHost(host : string) : void
- + getDBHost() : string
- + setDBDatabase(database : string) : void
- + getDBDatabase() : string
- + setDBIdent(ident : string) : void
- + getDBIdent() : string

Package: Employees

Employee

Attributes:

- - SSN : int
- - wage : double
- - password : string
- - type : int
- - lastName : string
- - firstName : string

Operations:

- + getSSN() : int
- + setSSN(ssn : int) : void
- + getWage() : double
- + setWage(wage : double) : void
- + getPassword() : string
- + setPassword(password : string) : void
- + getType() : int
- + setType(type : int) : void
- + getLastName() : string
- + setLastName(lastName : string) : void
- + getFirstName() : string
- + setFirstName(firstName : string) : void

Waiter (inherits from Employee)

Attributes: N/A

Operations:

- + takeOrder(order : Order) : void

Host (inherits from Employee)

Attributes: N/A

Operations:

- + assignTable(table : Table) : void
- + queueCustomer() : void

Busboy (inherits from Employee)

Attributes: N/A

Operations:

- + cleanTable(table : Table) : void

Cook (inherits from Employee)

Attributes: N/A

Operations:

- + takeOrder(order : Order) : void
- + deliverOrder(order : Order) : void

Manager (inherits from Employee)

Attributes: N/A

Operations:

- + assignTables() : void

Package: Busboy

BusboyDisplay

Attributes:

- - busboy : Busboy

Operations:

- + renderBusboy() : void

BusboyReadyControl

Attributes: N/A

Operations:

- + reportTables() : void

Package: Host

HostDisplay

Attributes:

- - host : Host

Operations:

- + renderHost() : void

HostAssignControl

Attributes: N/A

Operations:

- + assignTables() : void

HostQueueControl

Attributes: N/A

Operations:

- + queueCustomer() : void

Queue

Attributes:

- numInQueue : int

Operations:

- + queue() : void
- + dequeue() : void

Package: Cook

CookDisplay

Attributes:

- - cook : Cook

Operations: N/A

TakenOrdersDisplay (extends CookDisplay)

Attributes: N/A

Operations:

- + renderTakenOrders() : void

AvailableOrdersDisplay (extends CookDisplay)

Attributes: N/A

Operations:

- + renderAvailableOrders() : void

CompleteOrderControl

Attributes: N/A

Operations:

- + completeOrder(order : Order) : void

TakeOrderControl

Attributes: N/A

Operations:

- + takeOrder(order : Order) : void

Package: Waiter

WaiterMainDisplay

Attributes:

- - waiter : Waiter

Operations: N/A

OutOrdersDisplay (extends WaiterMainDisplay)

Attributes: N/A

Operations:

- + renderOutOrders() : void

TablesDisplay (extends WaiterMainDisplay)

Attributes: N/A

Operations:

- + renderTables() : void

WaiterTableDisplay

Attributes:

- - table : Table

Operations:

- + renderTable() : void

CloseTabControl

Attributes: N/A

Operations:

- + closeTab() : void

TableGoBackControl

Attributes: N/A

Operations:

- + goBack() : void

AddItemControl

Attributes: N/A

Operations:

- + addItem() : void

WaiterItemDisplay

Attributes:

- - table : Table
- - waiter: Waiter

Operations:

- + renderWaiterItem() : void

SelectItemControl

Attributes: N/A

Operations:

- + selectItem() : void

ItemGoBackControl

Attributes: N/A

Operations:

- + goBack() : void

Package: Manager

ManagerMainDisplay

Attributes:

- - manager : Manager

Operations:

- + renderManagerMain() : void

ManagerAssignControl

Attributes: N/A

Operations:

- + assignTables() : void

AddProfileControl

Attributes: N/A

Operations:

- + addProfile() : void

ManagerAddProfileDisplay

Attributes: N/A

Operations:

- + renderManagerAddProfile() : void

CreateProfileControl

Attributes: N/A

Operations:

- + createProfile() : void

EditProfileControl

Attributes: N/A

Operations:

- + editProfile() : void

ManagerSearchProfileDisplay

Attributes: N/A

Operations:

- + renderManagerSearchProfile() : void

SearchProfileControl

Attributes: N/A

Operations:

- + searchProfile() : void

ManagerSearchProfileResultsDisplay

Attributes: N/A

Operations:

- + renderSearchProfileResults() : void

SelectProfileControl

Attributes: N/A

Operations:

- + selectProfile() : void

ManagerEditProfileDisplay

Attributes: N/A

Operations:

- + renderEditProfile() : void

SaveChangesControl

Attributes: N/A

Operations:

- + saveChanges() : void

ModifyLayoutControl

Attributes: N/A

Operations:

- + openLayoutManager() : void

ManagerLayoutEditorDisplay

Attributes: N/A

Operations:

- + renderLayoutManager() : void

RedisgnControl

Attributes: N/A

Operations:

- + redesignLayout() : void

PayrollControl

Attributes: N/A

Operations:

- + openPayroll() : void

ManagerPayrollDisplay

Attributes: N/A

Operations:

- + renderPayroll() : void

ProcessFitlerControl

Attributes: N/A

Operations:

- + processFitler() : void

StatisticsControl

Attributes: N/A

Operations:

- + openStatistics() : void

ManagerStatisticsDisplay

Attributes: N/A

Operations:

- + renderStatistics() : void

Object Constraint Language (OCL) Contracts

```
context BusboyDisplay::renderBusboy(): void
      self.BusBoy -> true
      pre: n/a
      post: BusboyDisplay: renderBusboy()

context Busboy::reportTables(): void
      self.BusBoy -> true
      pre: n/a
      post: busboyReady: reportTables()


context TakenOrdersDisplay::renderTakenOrders(): void
      inv: self.Cook -> true
      pre: n/a
      post: dispTakenOrders: renderTakenOrders()

context AvailableOrdersDisplay::renderAvailableOrders(): void
      inv: self.Cook -> true
      pre: n/a
      post: dispAvailOrders: renderAvailableOrders()

context Cook::deliverOrder(order: Order): void
      inv: self.Cook -> true
      pre: setOrder: order = anOrder
      post: finishOrder: completOrder(order)

context Cook::takeOrder(order: Order): void
      inv: self.Cook -> true
      pre: setOrder: order = anOrder
      post orderTaken: takeOrder(order)


context Database::getQuery(queryString:string): AssociativeArray
      pre: qString: queryString = "string"
      post: qSent: getQuery(queryString)

context Database::processQuery(queryString: string): void
      pre: qString: queryString = "string"
      post: proStrg: processQuery(queryString)

context Database::setDBUser(user: string): void
      pre: setUser:     user = "user"
      post: setDBU:     setDBUser(user)

context Database::getDBUser(): string
      pre: n/a
      post: dbUserSent: getDBUser()

context Database::setDBPass(pass: string): void
      pre: setPass:     pass = "string"
      post: setDBP:     setDBPass(pass)

context Database::getDBPass(): string
      pre: n/a
```

```
      post: dbPassSent: getDBPass()

context Database::setDBHost(host: sring): void
      pre: setHost:     host = "host"
      post: setDBH:     setDBHost(host)

context Database::getDBHost(): void
      pre: n/a
      post: dbHostSent: getDBHost()

context Database::setDBDatabase(database: string): void
      pre: setDatabase: database = "database"
      post: setDBD:     setDBDatabase(database)

context Database::getDBDatabase(): string
      pre: n/a
      post dbDatabaseSent: getDBDatabase

context Database::setDBIdent(ident: string): void
      pre: setIdent:    ident = "ident"
      post setDBI:      setDBIdent(ident)

context Database::getDBIdent(): string
      pre: n/a
      post: dbIdentSent: getDBIdent()


context Employee::getSSN(): int
      inv: self.Employee.SSN -> aSSN
      pre: n/a
      post: SSNsent: getSSN()

context Employee::setSSN(ssn: int): void
      pre: assignSSN: SSN = aSSN
      post SSNassigned: getSSN(aSSN)

context Employee::getWage(): double
      inv: self.Employee.wage -> aWage
      pre: n/a
      post: wageSent: getWage()

context Employee::setWage(wage: double): void
      pre: assignWage: wage = aWage
      post wageAssigned: setWage(aWage)

context Employee::getPassword(): string
      inv: self.Employee.password -> "aPassword"
      pre: n/a
      post: passwordSent: getPassword()

context Employee::setPassword(password: string): void
      pre: assignPassword: password = "aPassword"
      post: passwordAssigned: setPassword("aPassword")

context Employee::getType(): int
      inv: self.Employee.type -> aType
      pre: n/a
```

```
      post: typeSent: getType()

context Employee::setType(type: int): void
      pre: assignType: type = aType
      post: typeAssigned: setType(aType)

context Employee::getLastName(): string
      inv: self.Employee.lastName -> "aLastName"
      pre: n/a
      post: lastNameSent: getLastName()

context Employee::setLastName(lastName: string): void
      pre: assignLastName: lastName = "aLastName"
      post: lastNameAssigned: setLastName("aLastName")

context Employee::getFirstName(): string
      inv: self.Employee.firstName -> "aFirstName"
      pre: n/a
      post: firstNameSent: getFirstName()

context Employee::setFirstName(firstName: string): void
      pre: assignFirstName: firstName = "aFirstName"
      post: firstNameAssigned: setFirstName("aFirstName")

context Cook::takeOrder(order: order): void
      inv: self.Waiter -> true
      pre: orderAssigned: order = aOrder
      post: orderTaken: takeOrder(aOrder)

context Host::assignTable(table: Table): woid
      inv: self.Host -> true
      pre: tableAssigned: table = aTable
      post: tblAssigned: assignTable(aTable)

context Host::queueCustomer(): void
      inv: self.Host -> true
      pre: n/a
      post: customerQueued: queueCustomer()

context Busboy::cleanTable(table: table): void
      inv: self.Busboy -> true
      pre: tableAssigned: table = aTable
      post: tableCleaned: cleanTable(aTable)

context cook::takeOrder(order: Order): void
      inv: self.Cook -> true
      pre: assignOrder: order = aOrder
      post: orderTaken: takeOrder(aOrder)

context Cook::deliverOrder(order: Order): void
      inv: self.Cook -> true
      pre: assignOrder: order = aOrder
      post: sendOrder: deliverOrder(aOrder)

context Manager::assignTables(): void
      inv: self.Manager -> true
      pre: n/a
```

```
      post: tablesAssigned: assignTable


context HostDisplay::renderHost(): void
      pre: isHost: Host = true
      post: hostDisplay: renderHost()

context Host::assignTables(): void
      inv: self.Host -> true
      pre: n/a
      post: tablesAssigned: assignTables

context Host::queueCustomer(): void
      inv: self.Host -> true
      pre: n/a
      post: customerQueued: queueCustomer()

context Queue::queue(): void
      inv: self.Host -> true
      pre: setNumInQueue: numInQueue = int
      post: queueobj: queue()

context Queue::dequeue(): void
      inv: self.Host -> true
      pre: setNumInQueue: numInQueue = int
      post: dequeuqobj: dequeue()


context ManagerMainDisplay::renderManagerMain(): void
      pre: isManager: Manager = true
      post: dispManagerWindow: renderManagerMain()

context Manager::assignTables(): void
      inv: self.Manager -> true
      pre: n/a
      post: tableAssignment: assignTables()

context Manager::addProfile(): void
      inv: self.Manager -> true
      pre: n/a
      post: profileAdded: addProfile()

context ManagerAddProfileDisplay::renderManagerAddProfile(): void
      inv: self.Manager -> true
      pre: n/a
      post: addProfileDisplayed: renderManagerAddProfile()

context CreateProfileControl::createProfile(): void
      inv: self.Manager -> true
      pre: n/a
      post: profileCreated:   createProfile()

context EditProfileControl::editProfile(): void
      inv: self.Manager -> true
      pre: n/a
      post: profileChanged:   editProfile()
```

```
context ManagerSearchProfileDisplay::renderManagerSearchProfile(): void
      inv: self.Manager -> true
      pre: n/a
      post dispManagerSearch: renderManagerSearchProfile()

context SearchProfileControl::searchProfile(): void
      inv: self.Manager -> true
      pre: n/a
      post: findProfile: searchProfile()

context
ManagerSearchProfileResultsControl::renderSearchProfileResults(): void
      inv: self.Manager -> true
      pre: n/a
      post: searchResultsDisp: renderSearchProfileResults()

context SelectProfileControl::selectProfile(): void
      inv: self.Manager -> true
      pre: n/a
      post: profileSelected: selectProfile()

context ManagerEditProfileDisplay::renderEditProfile(): void
      inv: self.Manager -> true
      pre: n/a
      post: editProfileWindowDisp: renderEditProfile()

context SaveChangesControl::saveChanges(): void
      inv: self.Manager -> true
      pre: n/a
      post: changesSaved: saveChanges()

context OpenLayoutManagerControl::openLayoutManager(): void
      inv: self.Manager -> true
      pre: n/a
      post: layoutWindowDisplayed: openLayoutManager()

context RedesignLayoutControl::redesignLayout(): void
      inv: self.Manager -> true
      pre: n/a
      post: layoutChanged: redesignLayout()

context OpenPayrollControl::openPayroll(): void
      inv: self.Manager -> true
      pre: n/a
      post: payrollAccessible: openPayroll()

context ManagerPayrollDisplay::renderPayroll(): void
      inv: self.Manager -> true
      pre: n/a
      post: payrollWindow: renderPayroll()

context ProcessFilterControl::processFilter(): void
      inv: self.Manager -> true
      pre: n/a
      post: filterOut: processFilter()

context OpenStatisticsControl::openStatistics(): void
```

```
      inv: self.Manager -> true
      pre: n/a
      post: accessStats: openStatistics()

context ManagerStatisticsDisplay::renderStatistics(): void
      inv: self.Manager -> true
      pre: n/a
      post: dispStats: renderStatistics()


context LoginDisplay::renderLogin():void
      pre: n/a
      post: loginDisplayed: renderLogin()

context LoginControl::authenticateUser(user: string, pass: string):
boolean
      pre: setUser: user = "aUser"
           setPassword: password = "aPassword"
      post: userAuthenticated: authentication("aUser", "aPassword")

context LogoutControl::confirmLogout(): boolean
      pre: n/a
      post: logout: confirmLogout()

context ClockControl::clockIn(): void
      pre: setUser: user = "aEmployee"
      post: userClocksIn: clockIn()

context Table::getTableID(): int
      inv: self.Utility.TableID -> aTableID
      pre: n/a
      post: TableIDsent: getTableID()

context Table::setWaiterID(waiterID: int): void
      pre: assignWaiterID: waiterID = aWaiterID
      post: waiterIDassigned: setWaiterID(aWaiterID)

context Table::getWaiterID(): int
      inv: self.Utility.waiterID -> aWaiterID
      pre: n/a
      post: waiterIDsent: getWaiterID()

context Table::setBusboyID(busboyID: int): void
      pre: assignBusboyID: busboyID = aBusboyID
      post: busboyIDassigned: setBusBoyID(aBusboyID)

context Table::getBusboyID(): int
      inv: self.Utility.busboyID -> aBusboyID
      pre: n/a
      post: busboyIDsent: getBusboyID()

context Table::setStatus(status: int): void
      pre: assignStatus: status = aStatus
      post: statusAssigned: setStatus(aStatus)

context Table::getStatus(): void
      inv: self.Utility.status -> aStatus
```

```
      pre: n/a
      post: statusSent: getStatus

context Order::setCookID(cookID: int): void
      pre: assignCookID: cookID = aCookID
      post: cookIDassigned: setCookID(aCookID)

context Order::getCookID(): int
      inv: self.Utility.cookID -> aCookID
      pre: n/a
      post: cookIDsent: getCookID()

context Order::setTableID(tableID: int): void
      pre: assignTableID: tableID = aTableID
      post: TableIDassigned: setTableID(aTableID)

context Order::setFoodID(foodID: int): void
      pre: assignFoodID: foodID = aFoodID
      post: foodIDassigned: setFoodID(aFoodID)

context Order::getFoodID(): int
      inv: self.Utility.foodID -> aFoodID
      pre: n/a
      post: foodIDsent: getFoodID()

context Order::getTimestamp(): MySQL timestamp
      inv: self.Utility.timeStamp -> MySQL
      pre: n/a
      post: timeStampSent: getTimestamp()

context Order::getOrderID(): int
      inv: self.Utility.orderID -> aOrderID
      pre: n/a
      post: orderIDsent: getOrderID()

context Food::getType(): int
      inv: self.Utility.type -> aType
      pre: n/a
      post: typeSent: getType()

context Food::setType(type: int): void
      pre: assignType: type = aType
      post: typeAssigned: setType(aType)

context Food::getName(): string
      inv: self.Utility.name -> "aName"
      pre: n/a
      post: nameSent: getName()

context Food::setName(name: string): void
      pre: assignName: name = "aName"
      post: nameAssigned: setName("aName")

context Food::getPrice(): double
      inv: self.Utility.price -> aPrice
      pre: n/a
      post: priceSent: getPrice()
```

```
context Food::setPrice(price: double): void
      pre: assignPrice: price = aPrice
      post: priceAssigned: setPrice(aPrice)


context OutOrdersDisplay::renderOutOrders(): void
      inv: self.Waiter -> true
      pre: n/a
      post: dispOutOrders: renderOutOrders()

context TablesDisplay::renderTables(): void
      inv: self.Waiter -> true
      pre: setTable: table = aTable
      post: dispWaiterTable: renderTable()

context CloseTabControl::closeTab(): void
      inv: self.Waiter -> true
      pre: n/a
      post: closeTabControl: closeTab()

context GoBackControl::goBack(): void
      inv: self.Waiter -> true
      pre: n/a
      post: goBackToTable: goBack()

context AddItemControl::addItem(): void
      inv: self.Waiter -> true
      pre: n/a
      post: newItem: addItem()

context SelectItemControl::renderWaiterItem(): void
      inv: self.Waiter -> true
      pre: setTable:  table = aTable
           setWaiter: waiter = aWaiter
      post: dispWaiterItem: renderWaiterItem()

context SelectItemControl::selectItem(): void
      inv: self.Waiter -> true
      pre: n/a
      post: pickAnItem: selectItem()

context GoBackControl::goBack(): void
      inv: self.Waiter -> true
      pre: n/a
      post: goBackToItem: goBack()
```

System Architecture and System Design

Architectural Styles

Repository

In the repository architectural style, subsystems access and modify a single data structure called the central **repository**. Subsystems are relatively independent and interact only through the repository. Control flow can be dictated either by the central repository or by the subsystems.

In our specific case, the **repository** is the relational database which stores all of our persistent data. The subsystems are the relatively independent host, water, cook, bus boy, and manager environments. These subsystems do not communicate to each other directly at any point in the program's lifecycle. Whatever interaction takes place between the aforementioned subsystems, takes place only though the changes in the state of the relational database. As far as the flow of control is concerned, it is dictated by the subsystems as they read from, and write to, the relational database.

Model/View/Controller

In the Model/View/Controller or MVC architectural style, subsystems are classified into three different types, **model** subsystems maintain domain knowledge, **view** subsystems display it to the user, and **controller** subsystems manage the sequence of interactions with the user.

In our particular case, the **model** subsystems is the relational database, the **view** subsystems are the dynamically generated HTML web pages that comprise the user interface, and the **control** subsystems are the PHP server side procedures (not necessarily procedures, essentially PHP code) which respond to and guide the overall systems' interaction with the user. As evident from this example, and is true in general, the MVC architecture is a more specific case of the **repository** architectural style.

Client/Server

In the **client/server** architectural style, a subsystem, the **server**, provides services to instances of other subsystems called the **clients**, which are responsible for interacting with the user.

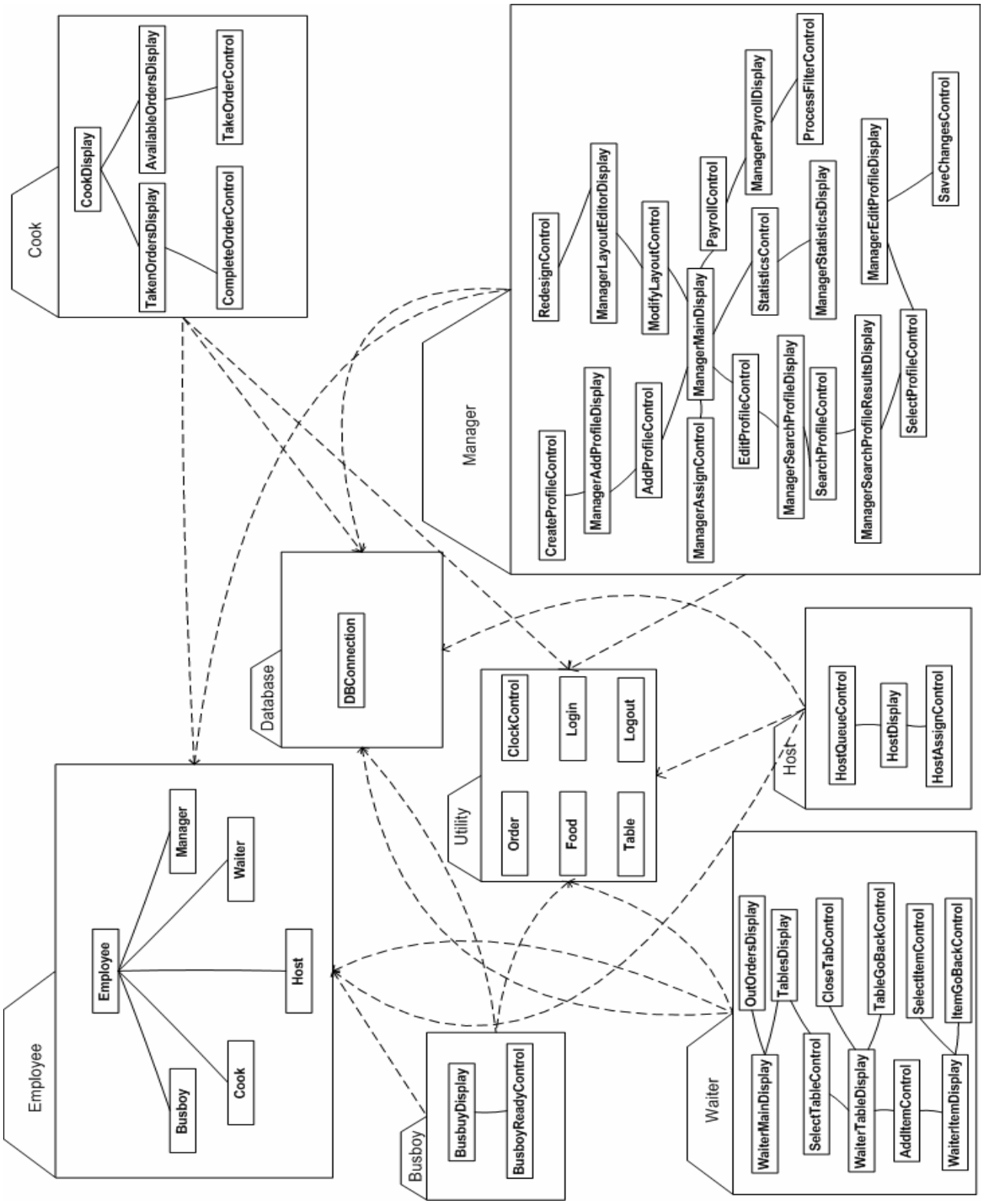
In the case of our system, the **server** is the machine running the relational database from which the **clients**, which are cook, bus boy, waiter, manager, and host, request the information they need. The **clients** also request to write information to the server. It is worth noting that the **server** is an outside system we are using, as the relational database is not built by us, hence it resides outside of the system.

Four-Tier

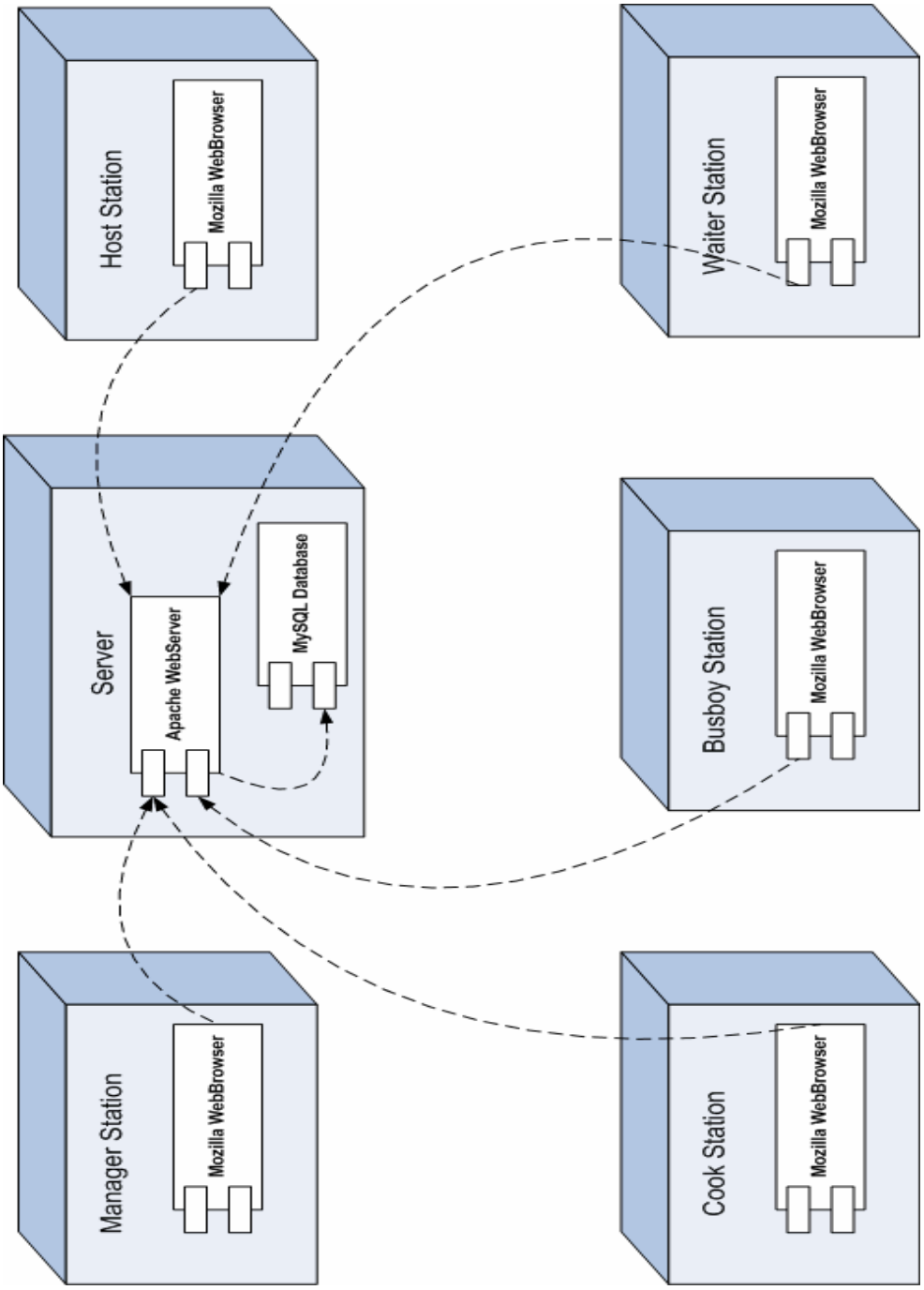
The **four-tier** architectural style is a common way to partition the overall system vertically. The four tiers are the **presentation client, presentation server, application logic, and storage.** All of the boundary objects that deal with the user are located in the **presentation client and presentation server** layers. The trick is that while boundary objects made custom to an interface client are not shared, those boundary objects common across diverse interfaces are shared. The **application layer** includes all control and entity objects, realizing the processing, rule checking, and notification required by the application. The **storage layer** realizes the storage, retrieval, and query of persistent objects.

For our system, the **presentation layer** includes the boundary objects custom to host, water, manager, bus boy, and cook such as their respective visual user interfaces. Boundary objects that are no different for all of cook, bus boy, manager, waiter, and host, make up the **presentation server.** An example of such an object would be the login screen which is identical for all users of the system. The **application layer** in our case is the PHP code governing the interaction between the system and its users. Finally, the **data storage** layer is expectantly the relational database.

Identifying Subsystems



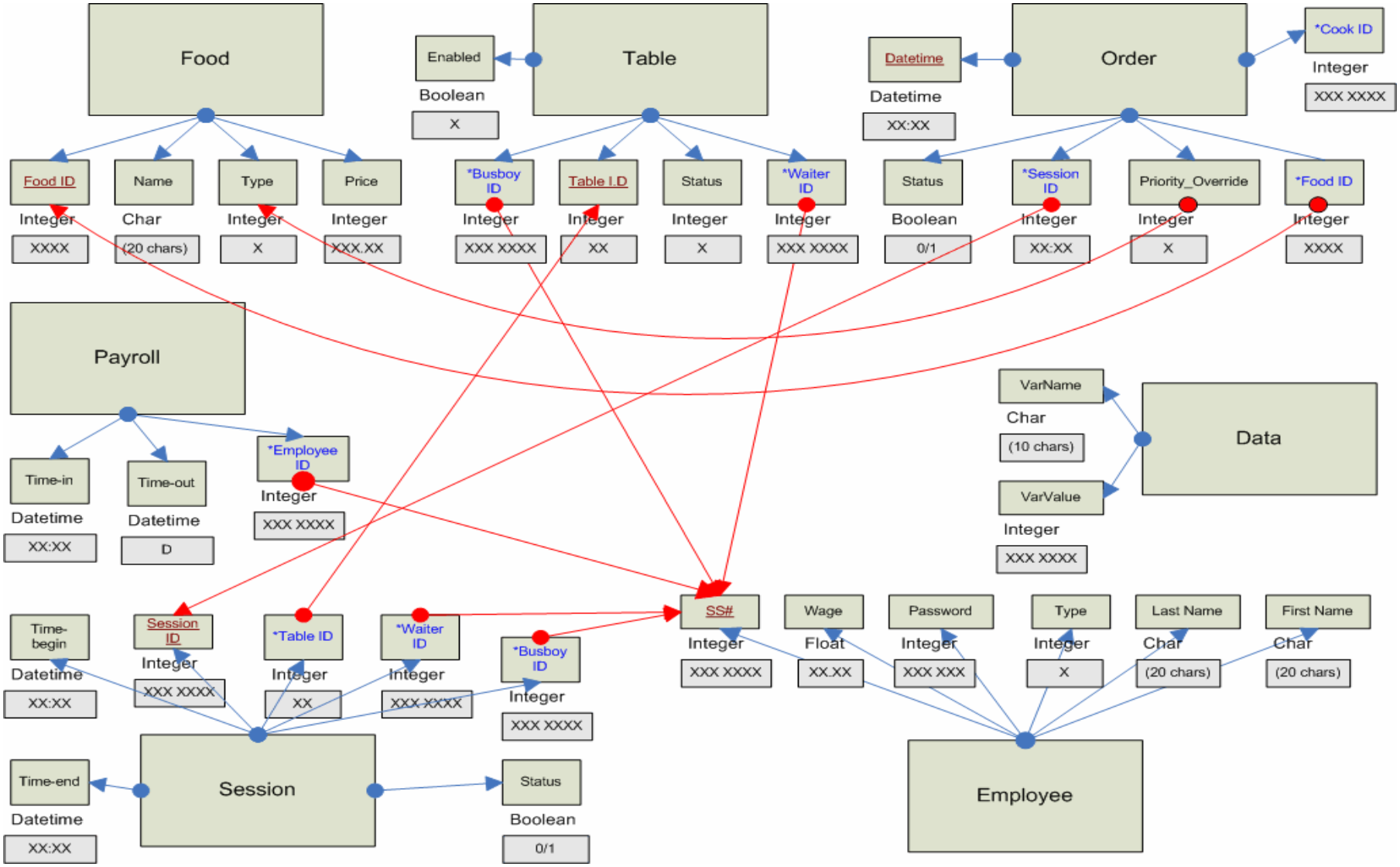
Mapping Subsystems to Hardware



Persistent Data Storage

During a restaurant's hours of operation there are torrents of information being communicated. Waiters adding orders, cooks completing orders, hosts seating people, are only a few of the actions which later entail other actor's participation. As such, information must span many sessions. In addition to these daily activities, information such as statistics and payroll must be logged and archived for future examination. All of this points to the need for a database which is both quick and simple, while providing good data abstraction. A Relational database, while not ideal, is the best available choice as it provides the correct balance of those attributes.

The diagram follows on the next page.


Food
Food ID
Integer
XXXX
Name
Char
(20 chars)
Type
Integer
X
Price
Integer
XXX.XX
Table
Enabled
Boolean
X
*Busboy ID
Integer
XXX XXXX
Table I.D
Integer
XX
Status
Integer
X
*Waiter ID
Integer
XXX XXXX
Order
Datetime
Datetime
XX:XX
*Cook ID
Integer
XXX XXXX
Status
Boolean
0/1
*Session ID
Integer
XX:XX
Priority_Override
Integer
X
*Food ID
Integer
XXXX
Payroll
Time-in
Datetime
XX:XX
Time-out
Datetime
D
*Employee ID
Integer
XXX XXXX
Data
VarName
Char
(10 chars)
VarValue
Integer
XXX XXXX
Time-begin
Datetime
XX:XX
Session ID
Integer
XXX XXXX
*Table ID
Integer
XX
*Waiter ID
Integer
XXX XXXX
*Busboy ID
Integer
XXX XXXX
SS#
Integer
XXX XXXX
Wage
Float
XX.XX
Password
Integer
XXX XXX
Type
Integer
X
Last Name
Char
(20 chars)
First Name
Char
(20 chars)
Time-end
Datetime
XX:XX
Session
Status
Boolean
0/1
Employee

Network Protocol

The system makes use of a web server (apache) running on the server, and a web browser (Mozilla) running on the client computers. As such it makes use of http. This option was especially attractive as it allows for great flexibility in design and high reliability (http is the most used protocol on the internet)

PHP, The Server Side Scripting Language chosen allows for seamless connection to a MySQL database server using the PHP-MySQL connection extension. The PHP-MySQL extension makes use of the MySQL Client/Server Protocol. This is an immensely powerful protocol and will facilitate all the server to database server connections. Aside from being highly robust, MySQL is completely free (GPL) and enables us to invest resources in other facets of the system.

Global Control Flow

Execution Order

At every page the user is presented with multiple controls which he/she may use to initiate a given action, thus allowing for easy and efficient navigation of the system. This method of control is known as event-driven control.

Time Dependency

While the system does not have any “true” time dependency, since a user interface is evolved, responses should be met in a timely fashion. After some research, the following data has been compiled.

- 100 Milliseconds is about the limit for having the user feel that the system is reacting instantaneously, meaning that no special feedback is necessary except to display the result.
- 1000 Milliseconds is about the limit for the user's flow of thought to stay uninterrupted, even though the user will notice the delay. Normally, no special feedback is necessary during delays of more than 100 milliseconds but less than 1000 milliseconds, but the user does lose the feeling of operating directly on the data.

Our goal is to respond to the user within 100 Milliseconds. However, if this is not met, 1000 Milliseconds will be considered an acceptable time.

Concurrency

Since our system is implemented via a server-side scripting language, it is, by nature, multithreaded. This is completely seamless to us (the programmer) and saves a lot of development time while making use of proven technologies. Apache, PHP, MySQL, are all proven to be solid, enterprise caliber software. These are multithreaded and allow for many concurrent users and concurrent database queries.

Hardware Requirements

Server

- 2.0 GHz CPU
- 1GBytes RAM
- 10/100/1000 NIC
- 10 Gbytes hard disk space

Client Stations

- 1.6 GHz CPU
- 512MBytes RAM
- 10/100/1000 NIC
- 17” Color, Touchscreen LCD with native resolution of atleast 1024 x 768 pixels
- 1 Gbytes hard disk space
- GAOTek 2407 RFID PROXPOINT Card Reader

Network

The system makes use of standard 10/100/1000 Ethernet as an intranet to facilitate communications between the clients and server. Any 10/100/1000 switch may be used to connect the system. Wireless Ethernet (802.11g) may be used if Ethernet wiring is unavailable or cannot be installed.

Algorithms and Data Structures

Algorithms

The Shul'chan Algorithm

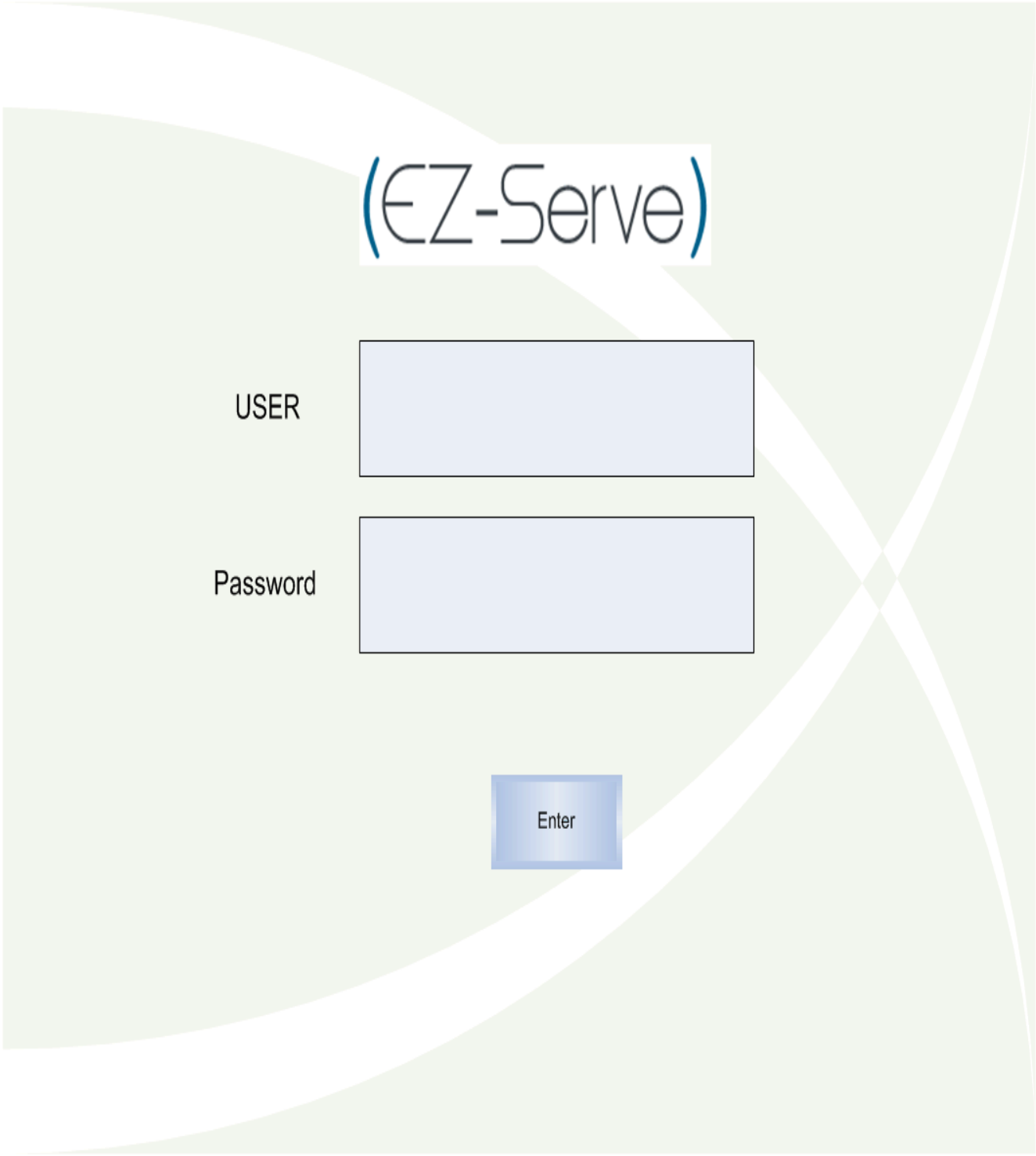
The Shul'chan (or table, in Hebrew) algorithm allows for easy manipulation of a restaurant layout. The table arrangement algorithm first requests approximate x by y dimensions of the array. It then shows the user an x by y array of tables The User may then go through each cell in the array and select it to be blank or contain a table.

Data Structures

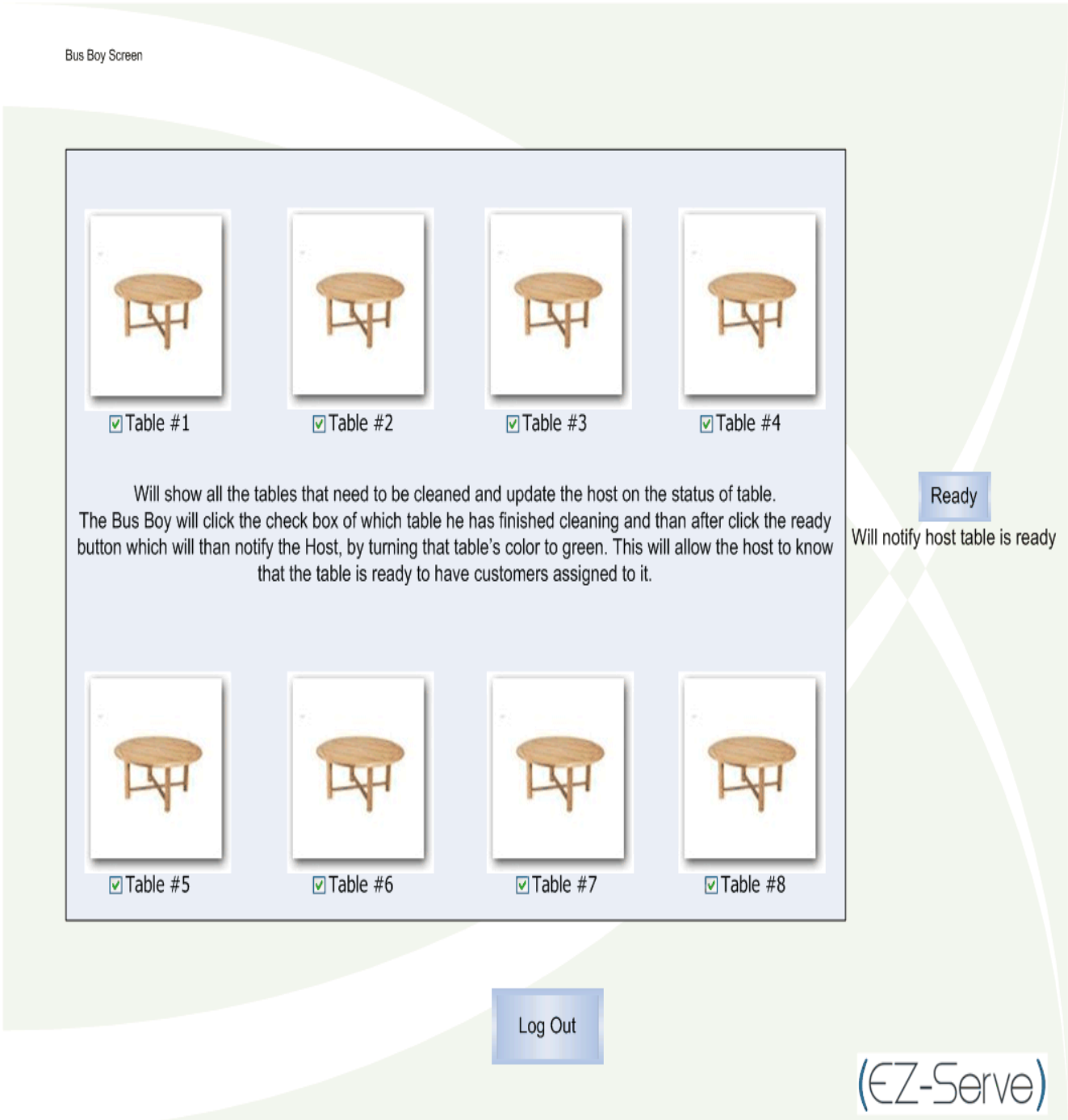
I have chosen the queue data structure to represent the customers waiting to be seated in the event of a full house the following reasons. First of all, this data structure is an organically good fit for the job at hand, as the customers do in reality form a queue when they await to be seated. That is the case because just like in real life, the queue queues the elements on its end, and dequeues the elements from its beginning. Secondly, for the type of operations needed for our purposes, queue offers the best performance. It is very fast at adding elements to its end and removing elements from its beginning. The fact that random access afforded to us by a data structure such as an Array or a Linked List is not possible is not relevant, since we do not need to access the data randomly. We also do not mind the fact the queue has a limited size, and cannot like the Linked List expand as much as we want it to. This is the case because there is only so many people that we can physically advise to stick around, and this number is in fact the size of our queue. As evident from this discussion, the queue is without a doubt the best possible data structure for our needs.

User Interface Design and Implementation

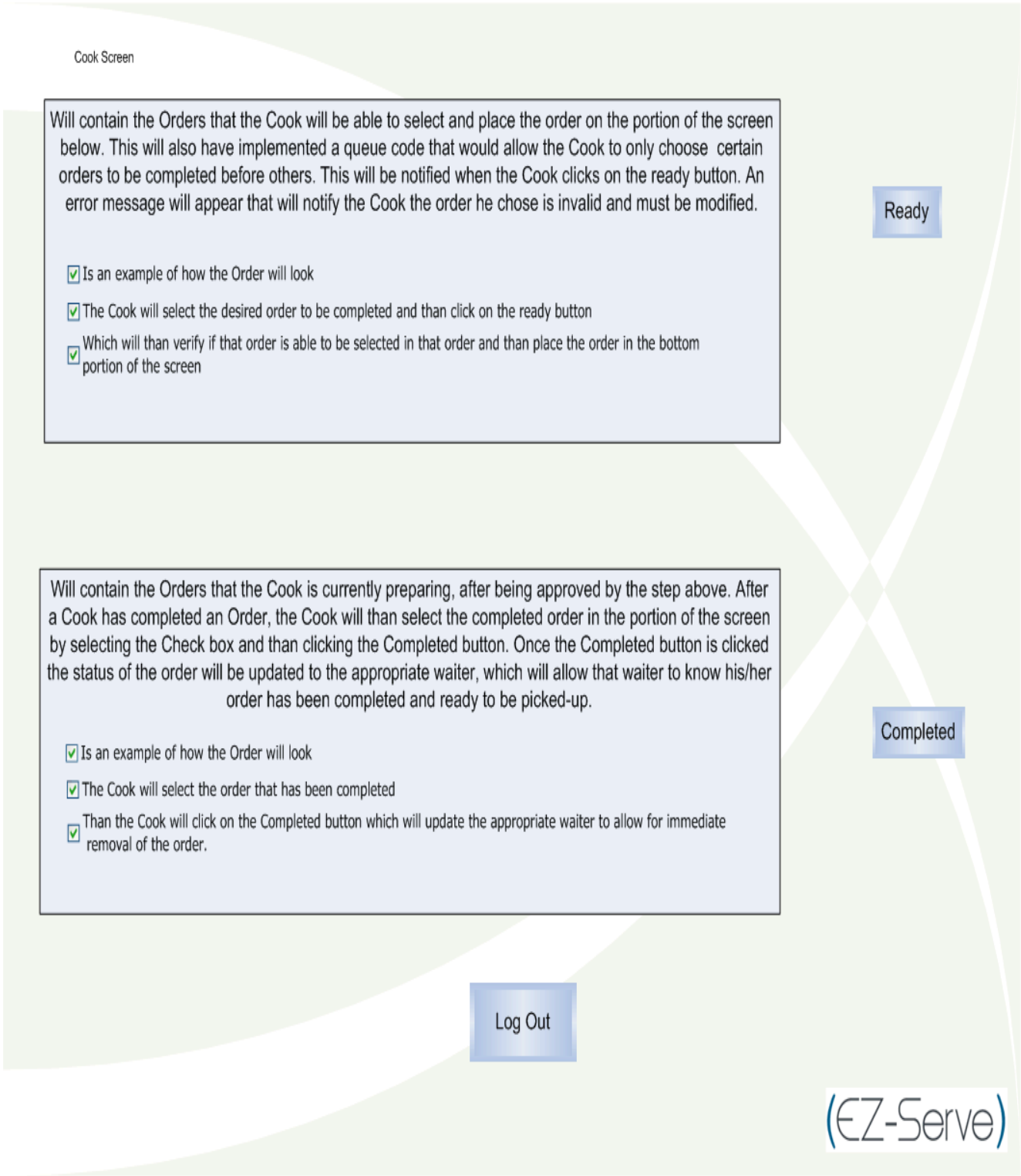
Login Screen



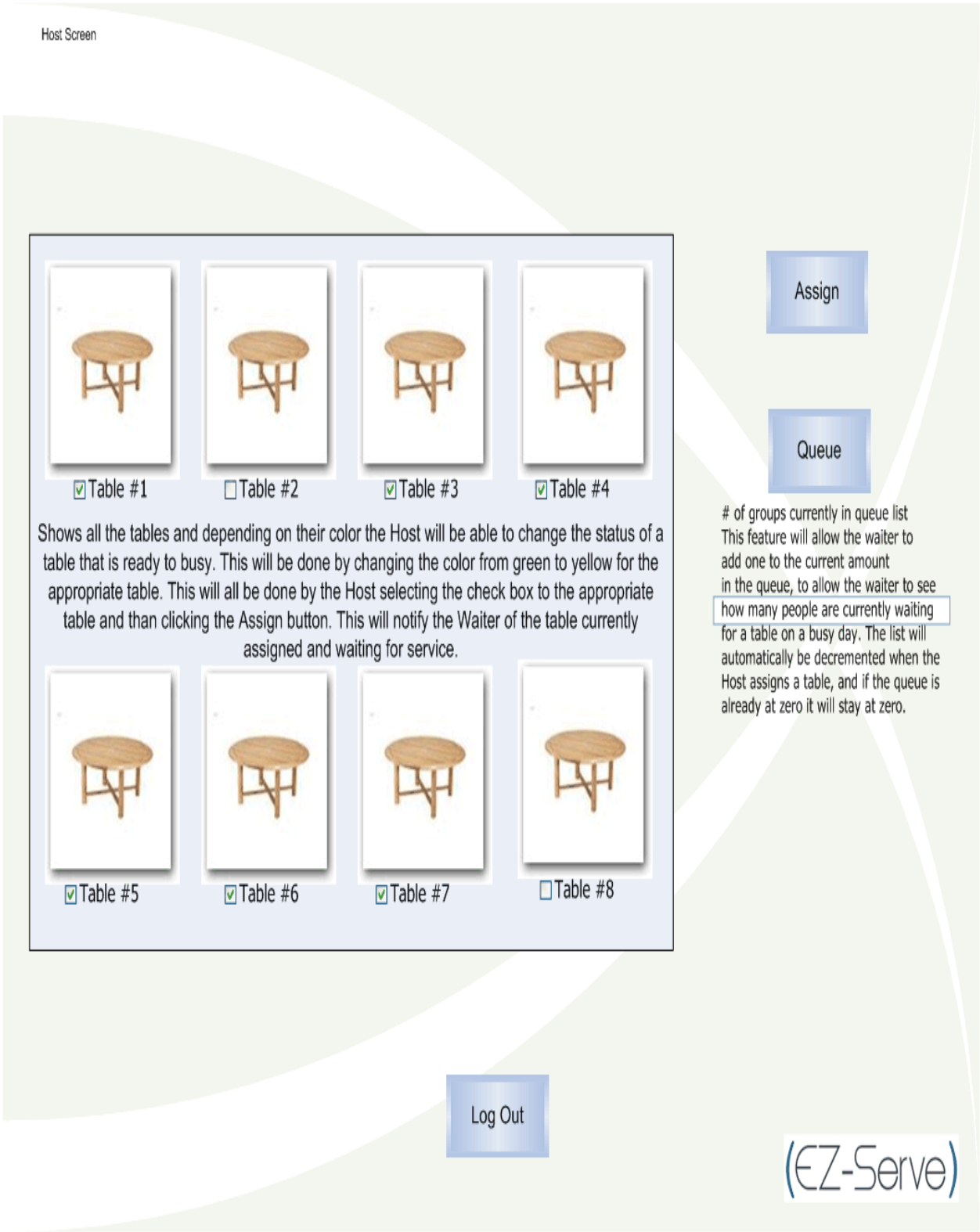
Busboy Screen



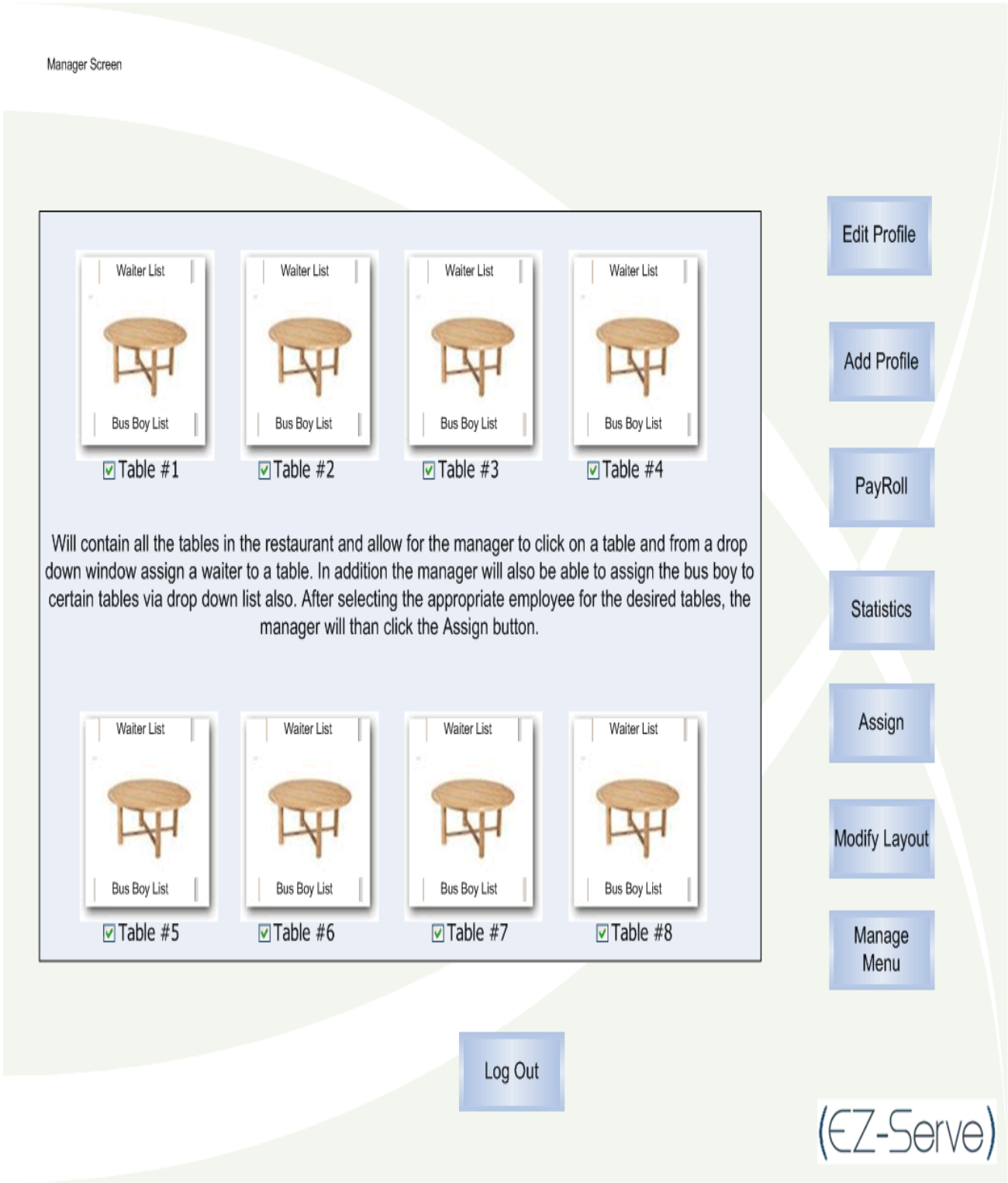
Cook Screen


Cook Screen
Will contain the Orders that the Cook will be able to select and place the order on the portion of the screen below. This will also have implemented a queue code that would allow the Cook to only choose certain orders to be completed before others. This will be notified when the Cook clicks on the ready button. An error message will appear that will notify the Cook the order he chose is invalid and must be modified.
Is an example of how the Order will look
The Cook will select the desired order to be completed and than click on the ready button
Which will than verify if that order is able to be selected in that order and than place the order in the bottom portion of the screen
Ready
Will contain the Orders that the Cook is currently preparing, after being approved by the step above. After a Cook has completed an Order, the Cook will than select the completed order in the portion of the screen by selecting the Check box and than clicking the Completed button. Once the Completed button is clicked the status of the order will be updated to the appropriate waiter, which will allow that waiter to know his/her order has been completed and ready to be picked-up.
Is an example of how the Order will look
The Cook will select the order that has been completed
Than the Cook will click on the Completed button which will update the appropriate waiter to allow for immediate removal of the order.
Completed
Log Out
(EZ-Serve)

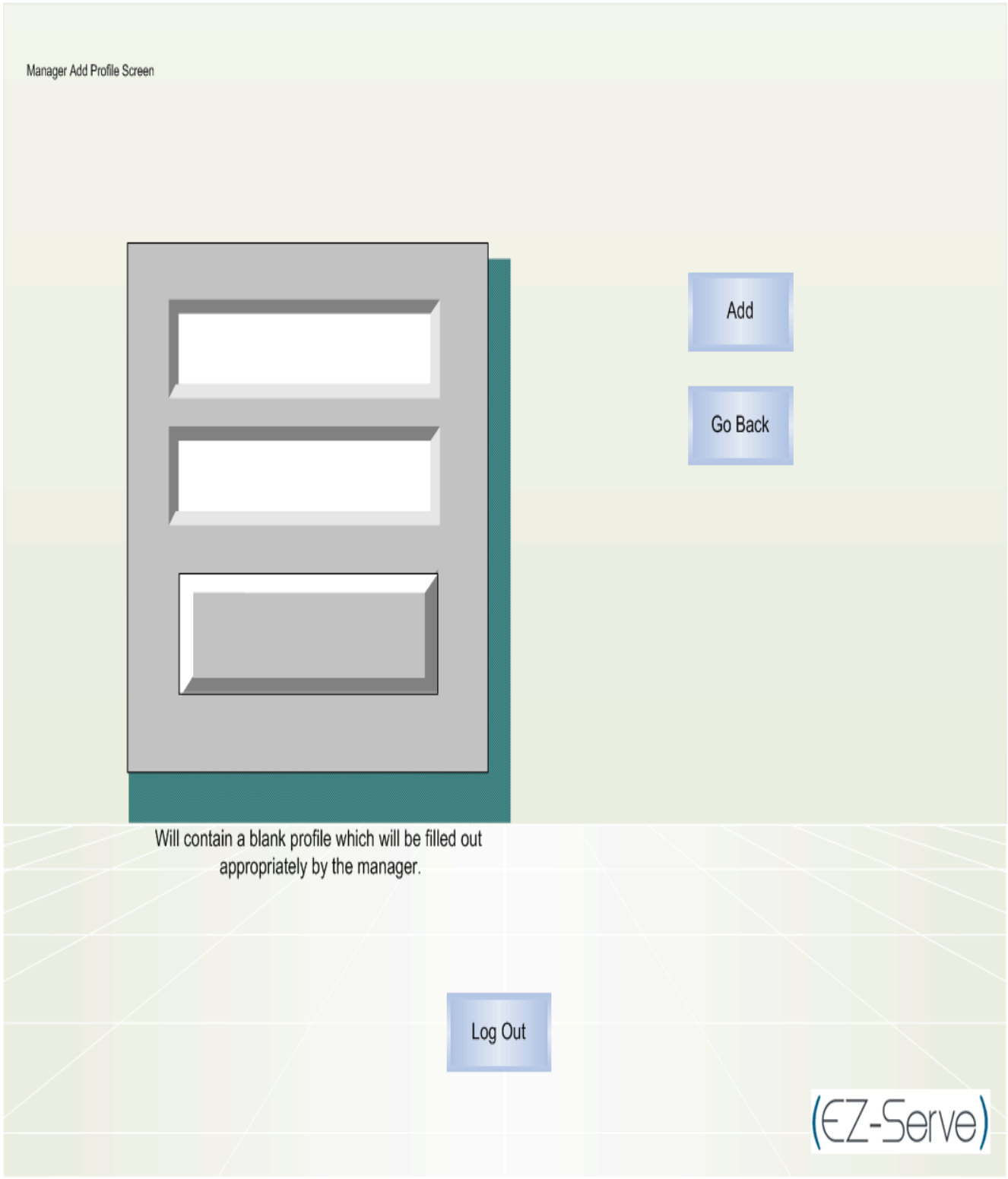
Host Screen


Host Screen
Table #1
Table #2
Table #3
Table #4
Shows all the tables and depending on their color the Host will be able to change the status of a table that is ready to busy. This will be done by changing the color from green to yellow for the appropriate table. This will all be done by the Host selecting the check box to the appropriate table and than clicking the Assign button. This will notify the Waiter of the table currently assigned and waiting for service.
Table #5
Table #6
Table #7
Table #8
Assign
Queue
# of groups currently in queue list
This feature will allow the waiter to add one to the current amount in the queue, to allow the waiter to see how many people are currently waiting for a table on a busy day. The list will automatically be decremented when the Host assigns a table, and if the queue is already at zero it will stay at zero.
Log Out
(EZ-Serve)

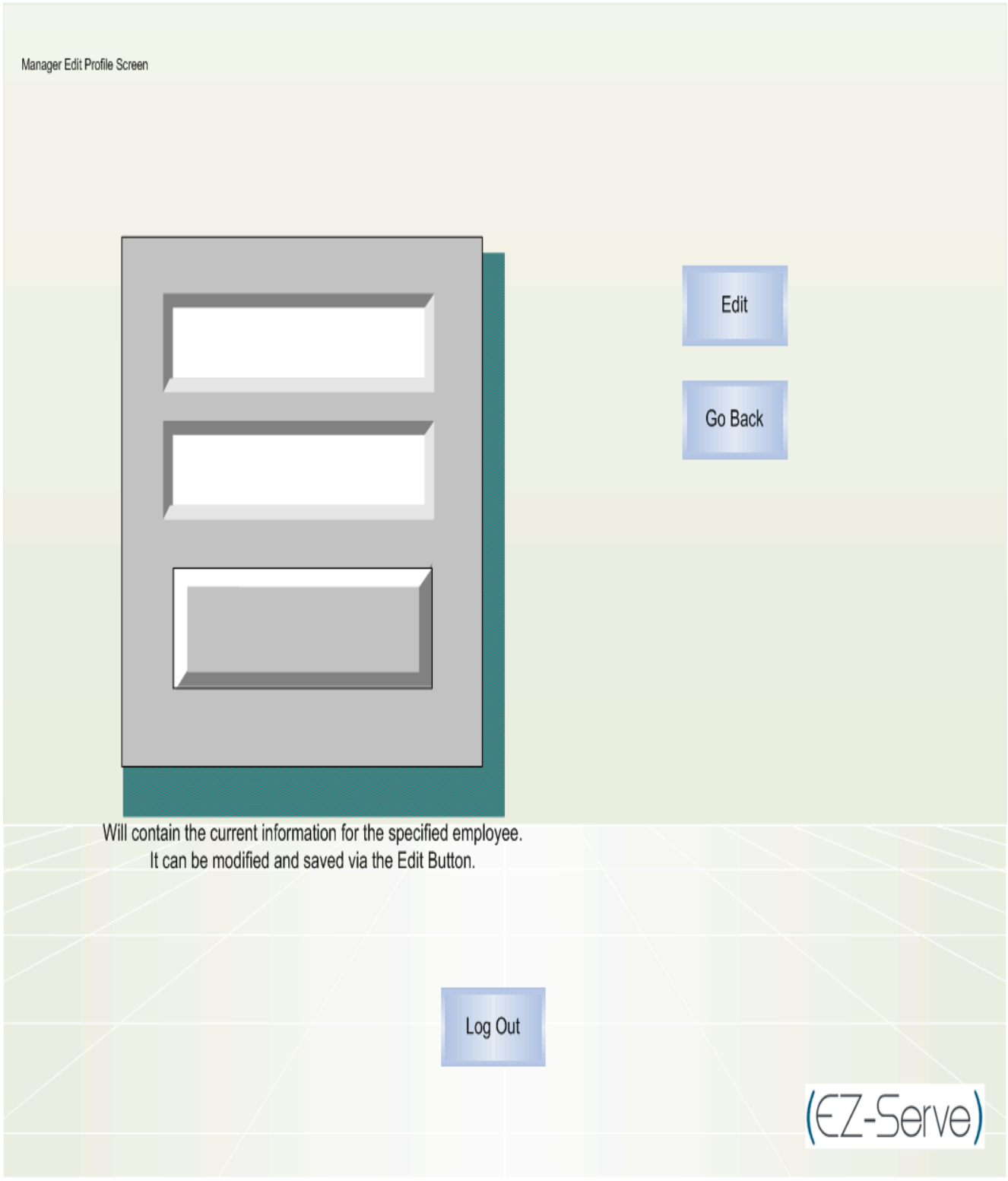
Manager Main Screen



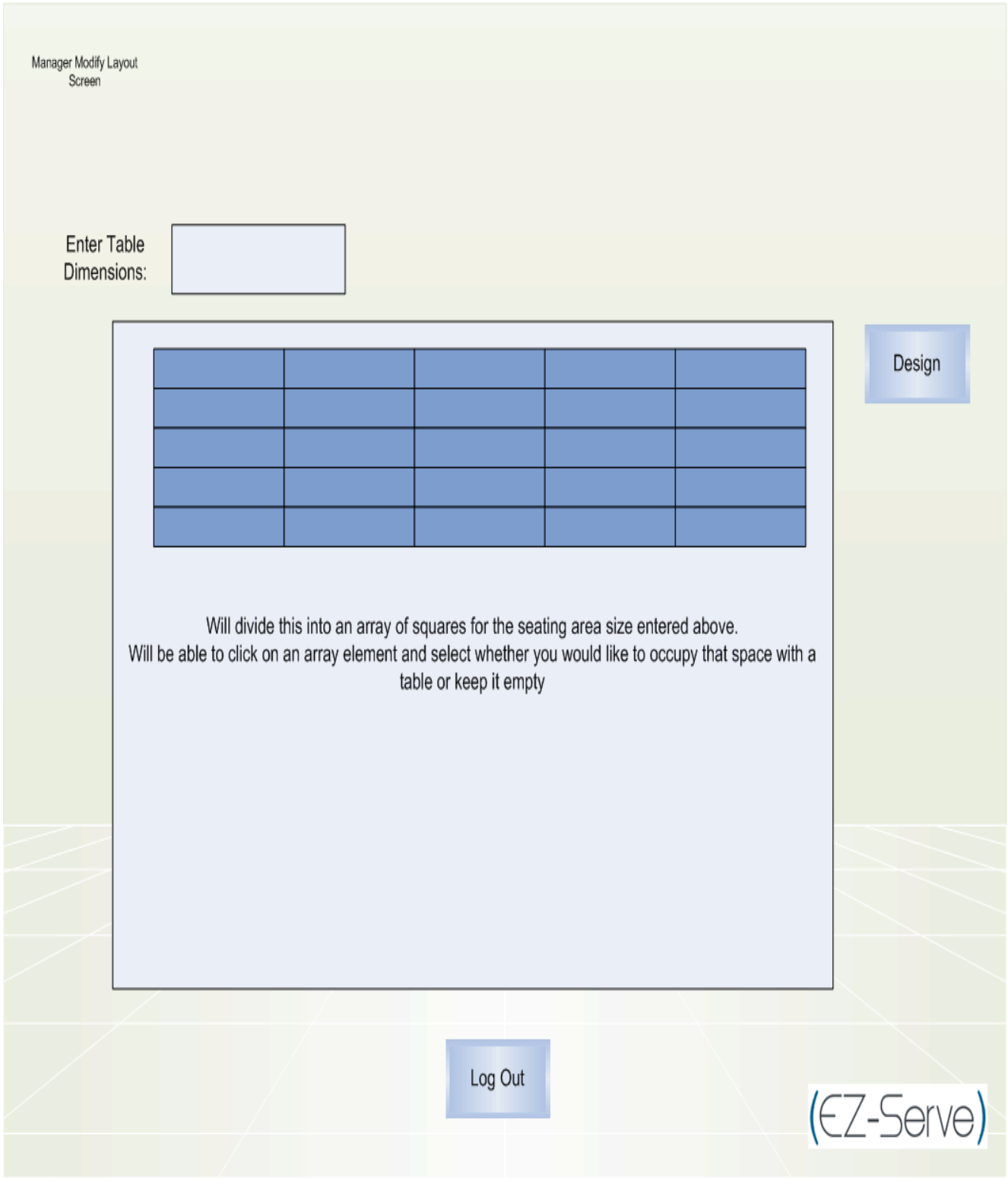
Manager Add Screen



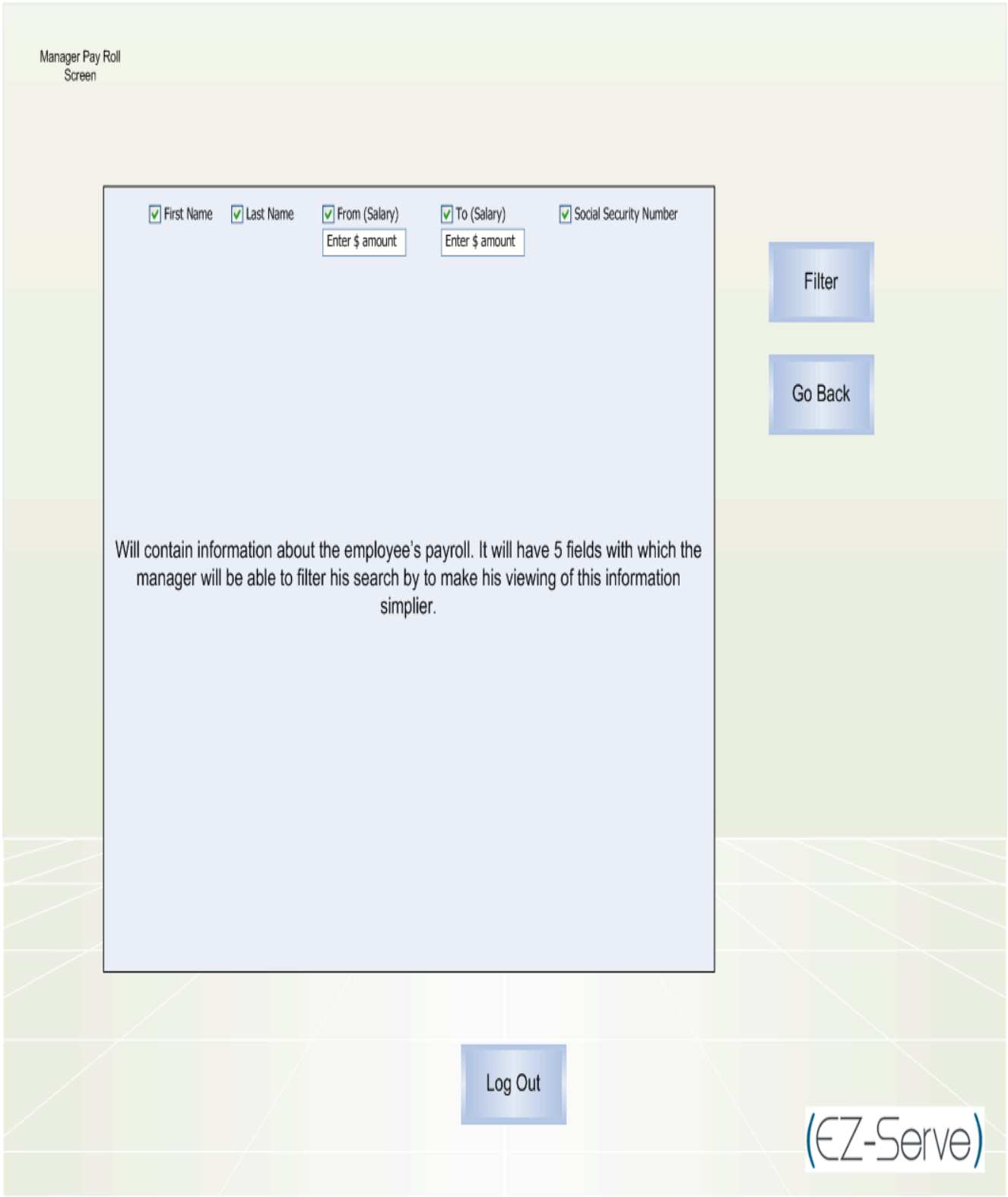
Manager Edit Profile Screen



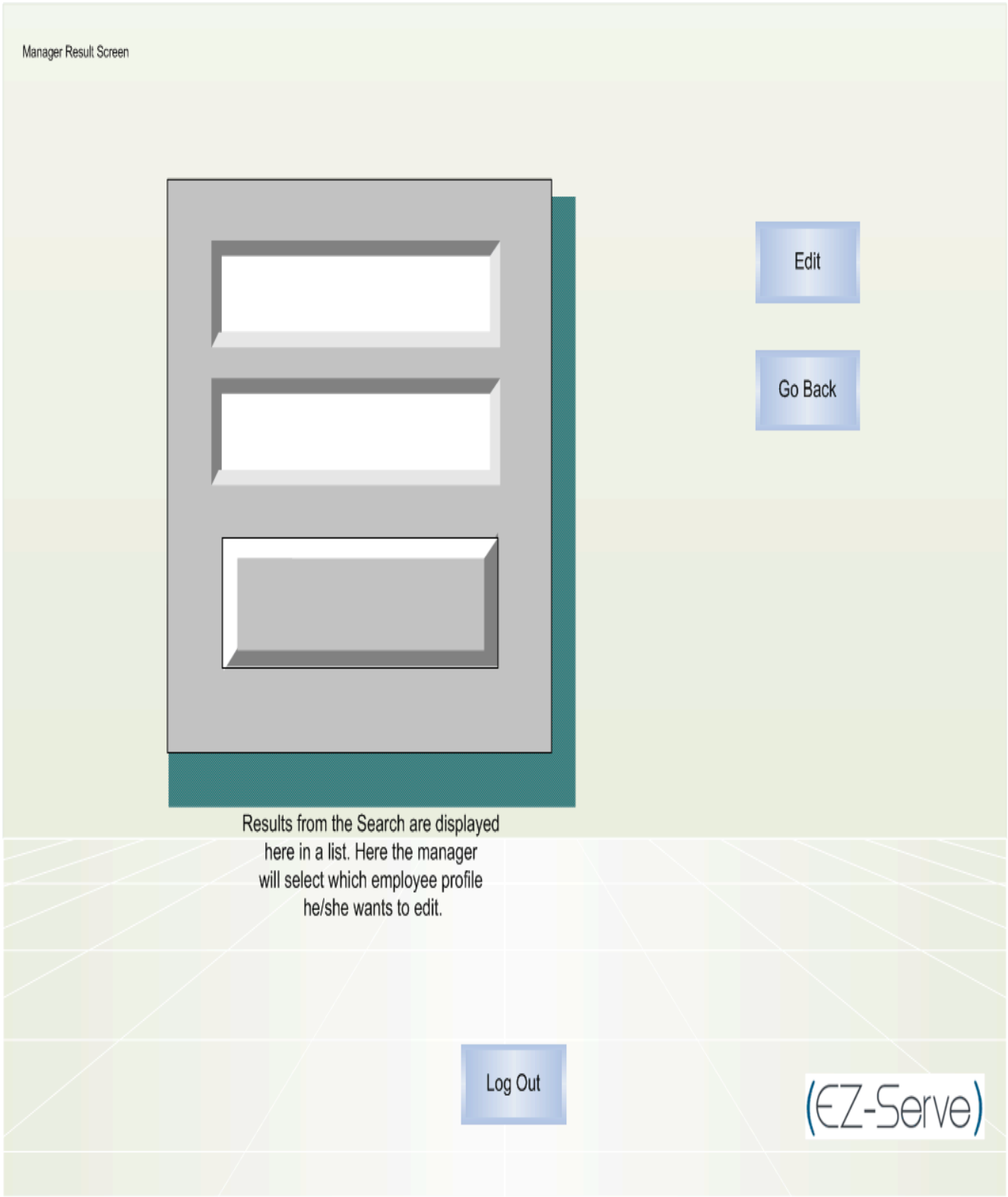
Manager Modify Layout Screen



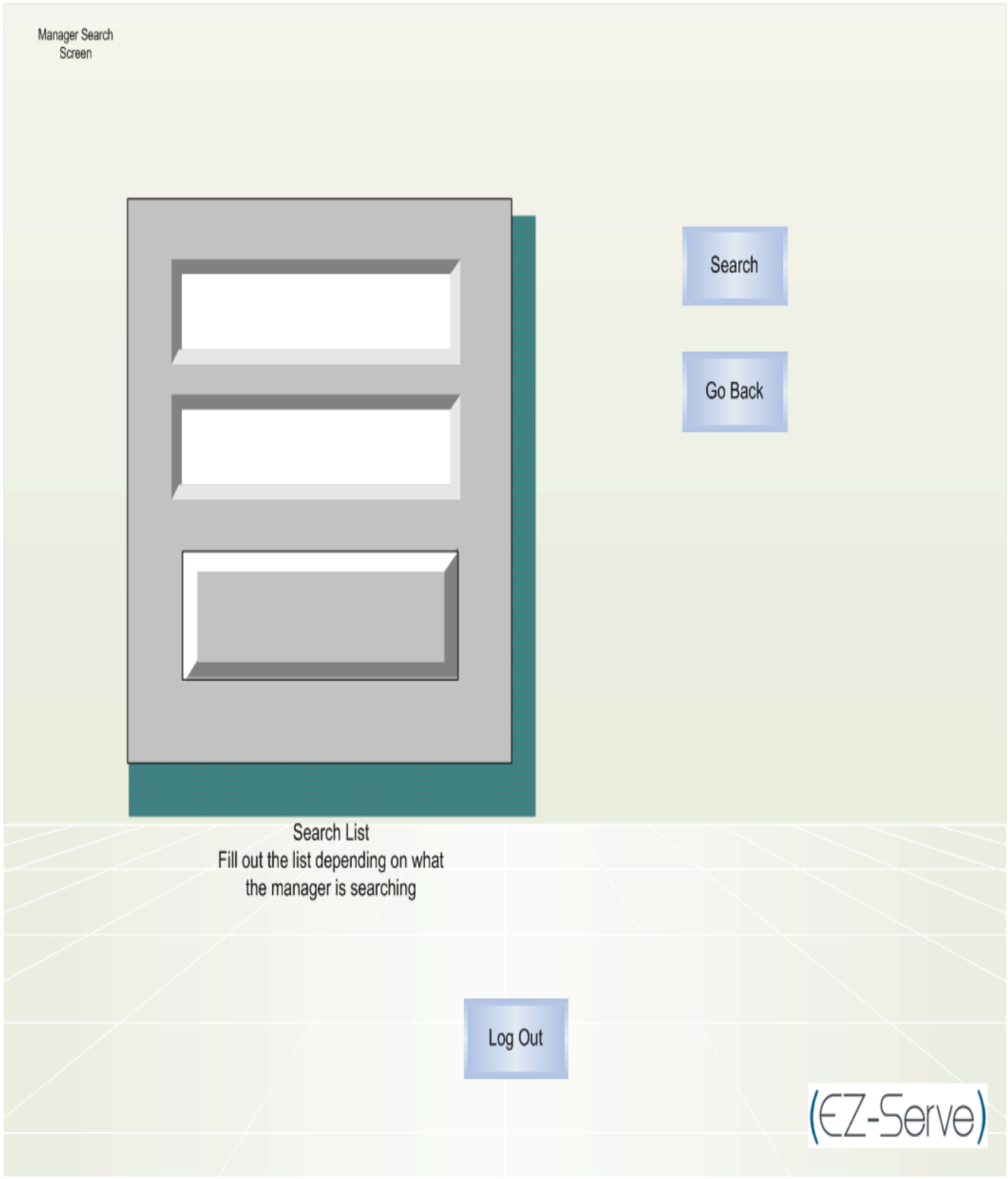
Manager Payroll Screen



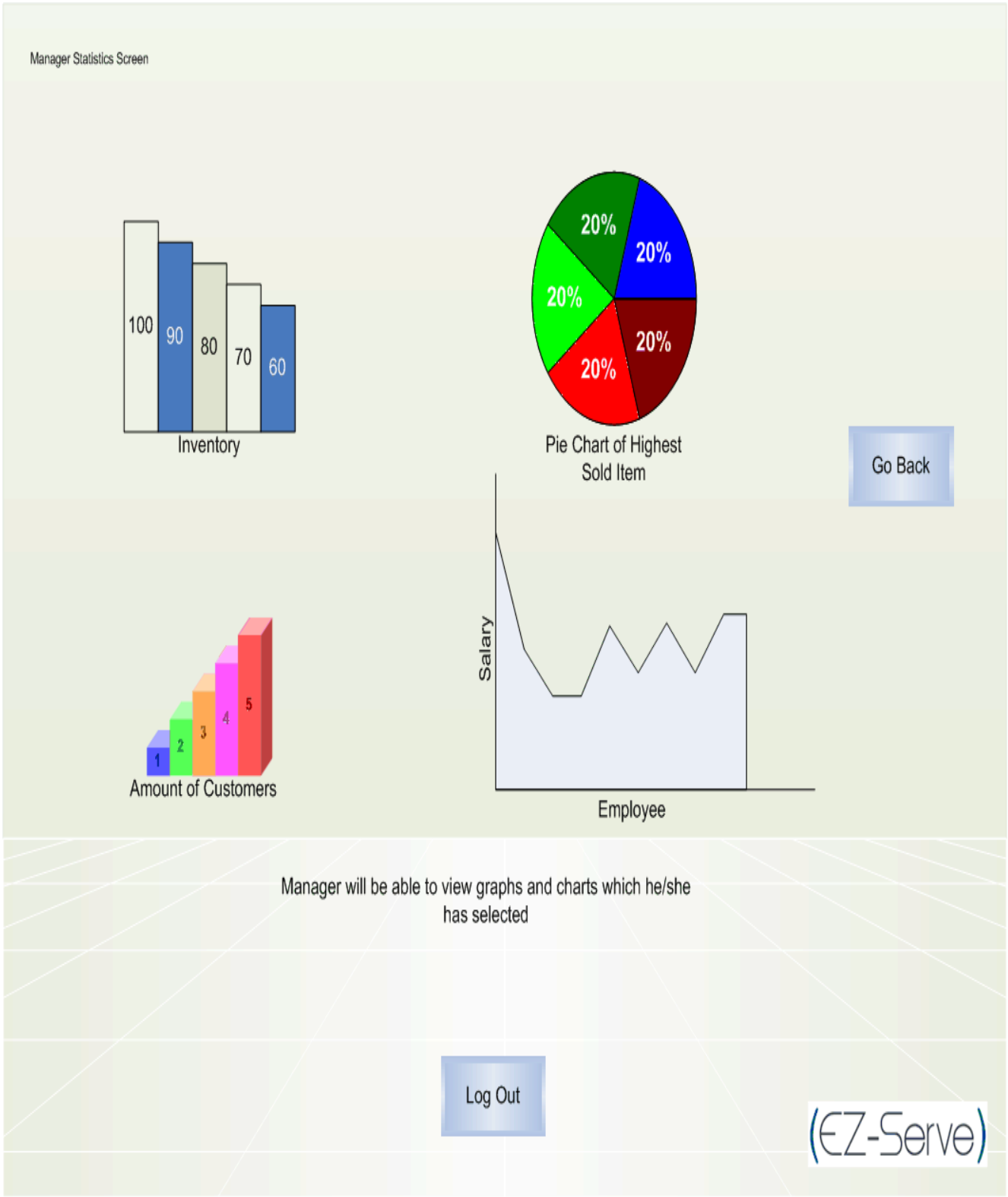
Manager Result Screen



Manager Search Screen

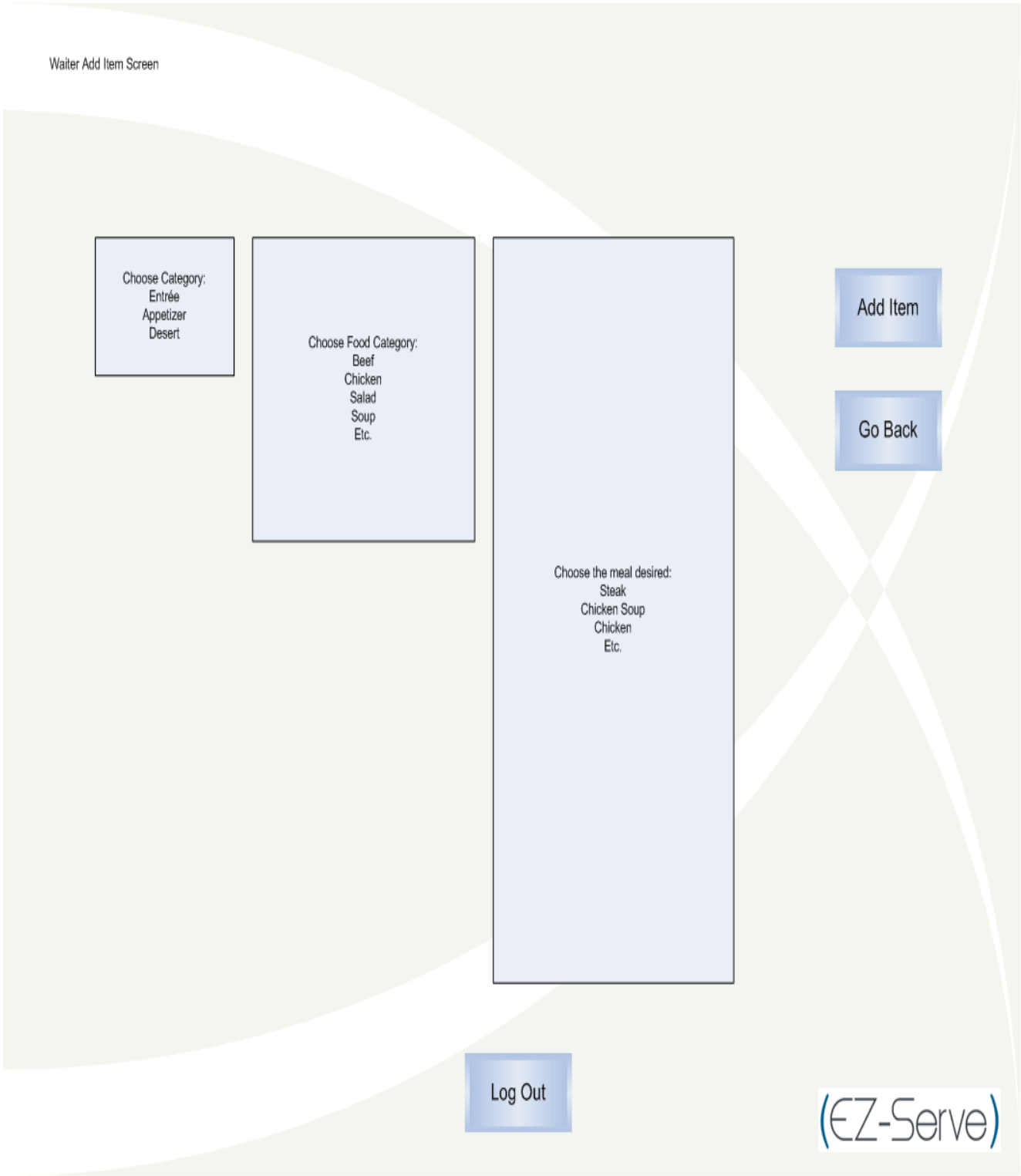


Manager Statistics Screen

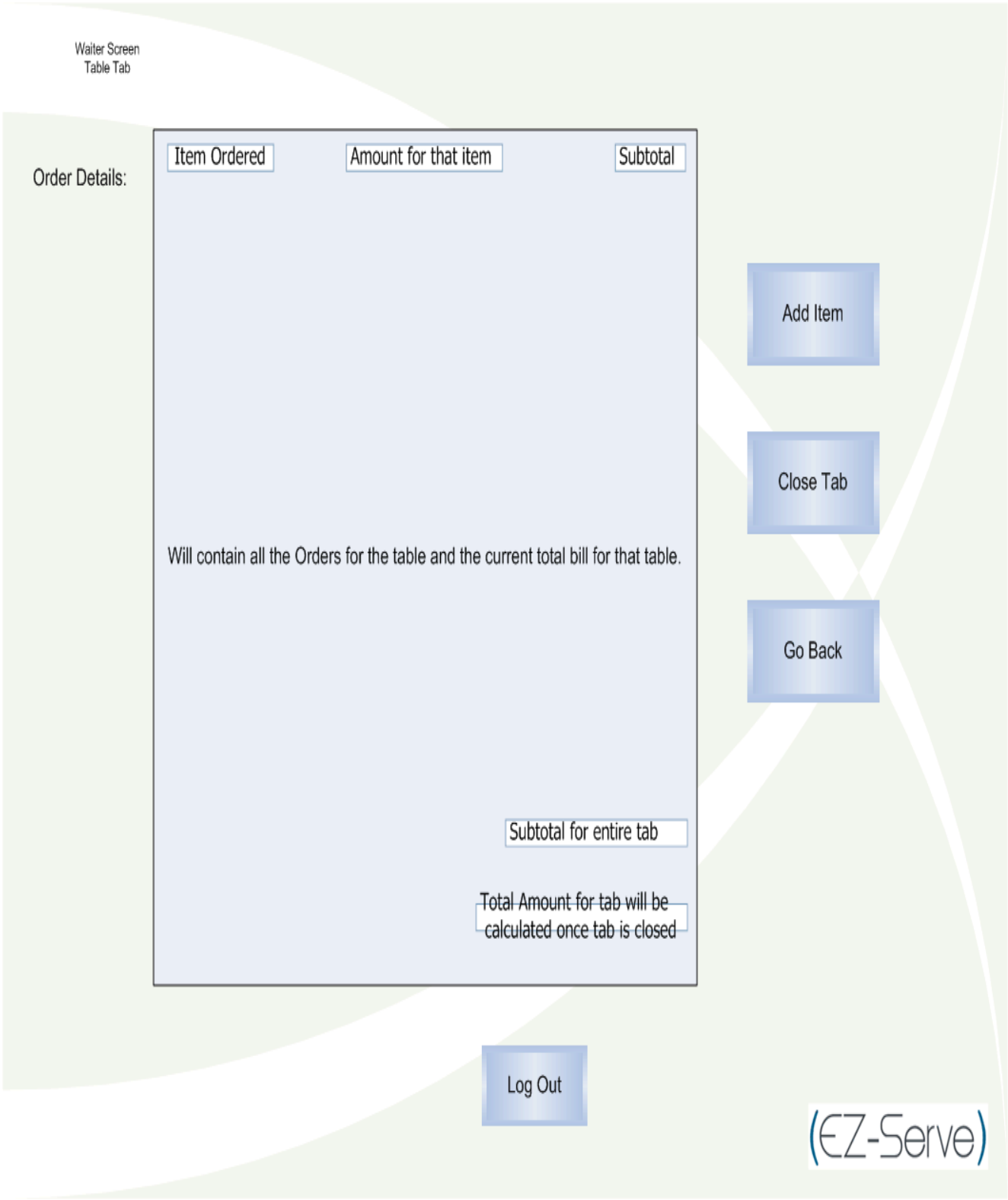


Waiter Main Screen

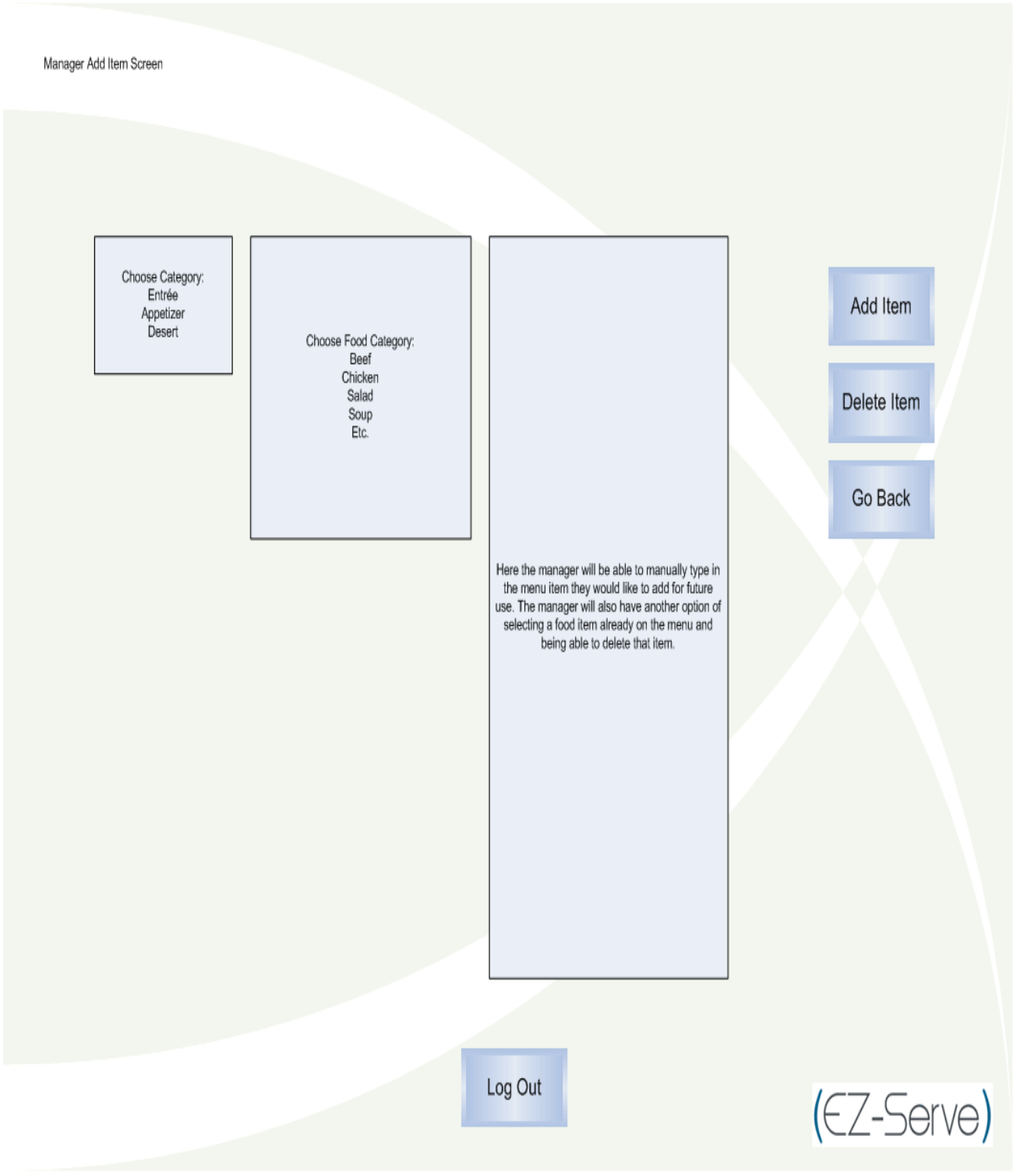
Waiter Add Item Screen



Waiter Tab Screen



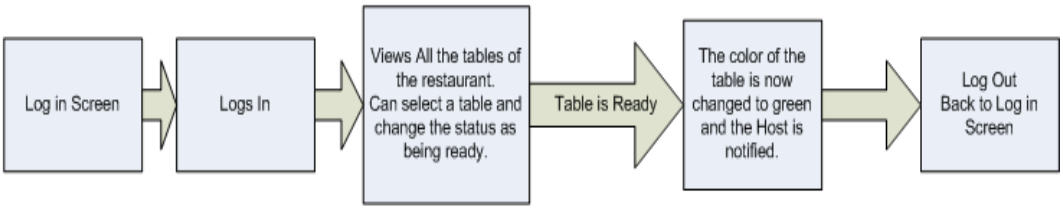
Manager Manage Menu Screen



User Effort Estimation

Bus Boy

Possible Paths for the Bus Boy:



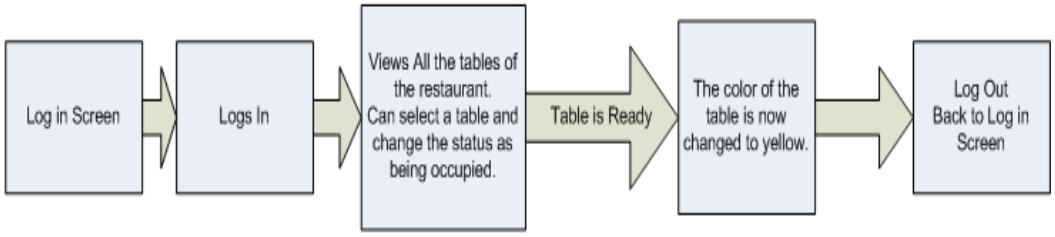
Total amount of clicks for the bus boy to:

-Notify Host of ready table:

1 Click Log in (Clerical data entry)
1 Click Select Table (User-interface navigational data entry)
1 Click Table Ready (User-interface navigational data entry)
1 Click Log out (User-interface navigational data entry)
4 Clicks Total
¾ (User-interface navigational data entry) ¼ (Clerical data entry)

Host

Possible Paths for the Host:



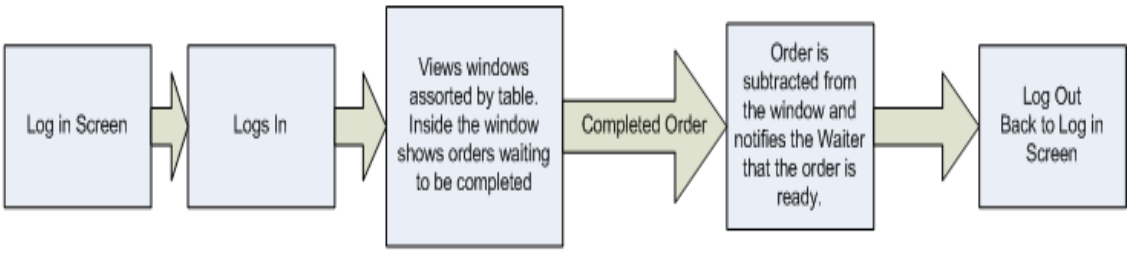
Total amount of clicks for the Host to:

-Change status of table:

1 Click Log in (Clerical data entry)
1 Click Select Table (User-interface navigational data entry)
1 Click Table Occupied (User-interface navigational data entry)
1 Click Log out (User-interface navigational data entry)
4 Clicks Total
¾ (User-interface navigational data entry) ¼ (Clerical data entry)

Cook

Possible Paths for the Cook:



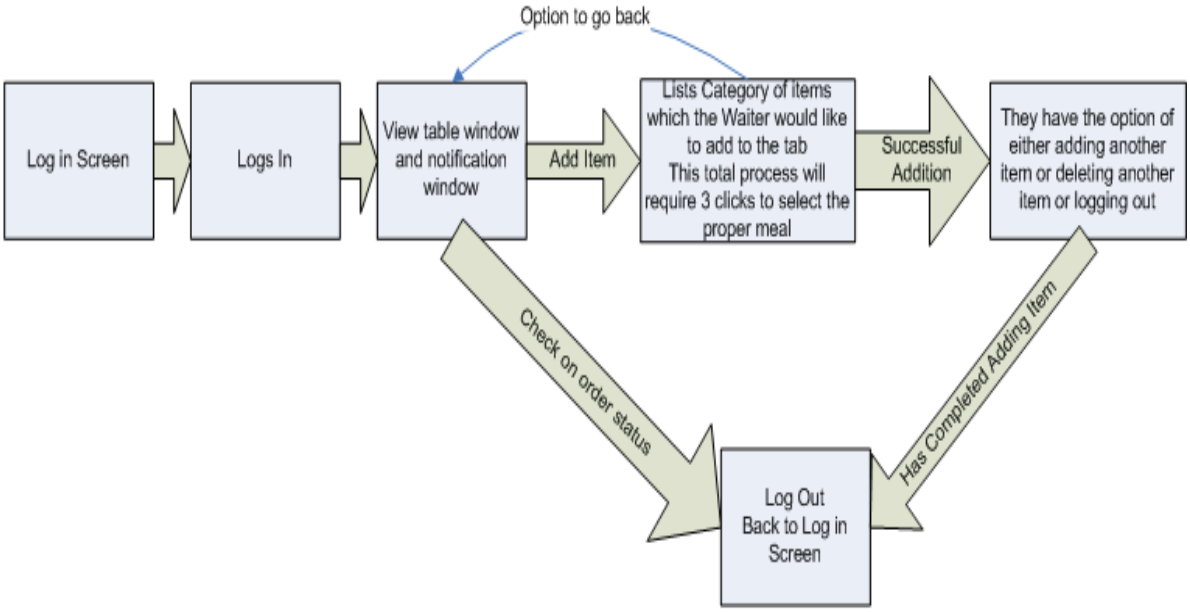
Total amount of clicks for the cook to:

-Notify Waiter of completed order:

1 Click Log in (Clerical data entry)
1 Click Select Order (User-interface navigational data entry)
1 Click Order Ready (User-interface navigational data entry)
1 Click Log out (User-interface navigational data entry)
4 Clicks Total
¾ (User-interface navigational data entry) ¼ (Clerical data entry)

Waiter

Possible Paths for the Waiter



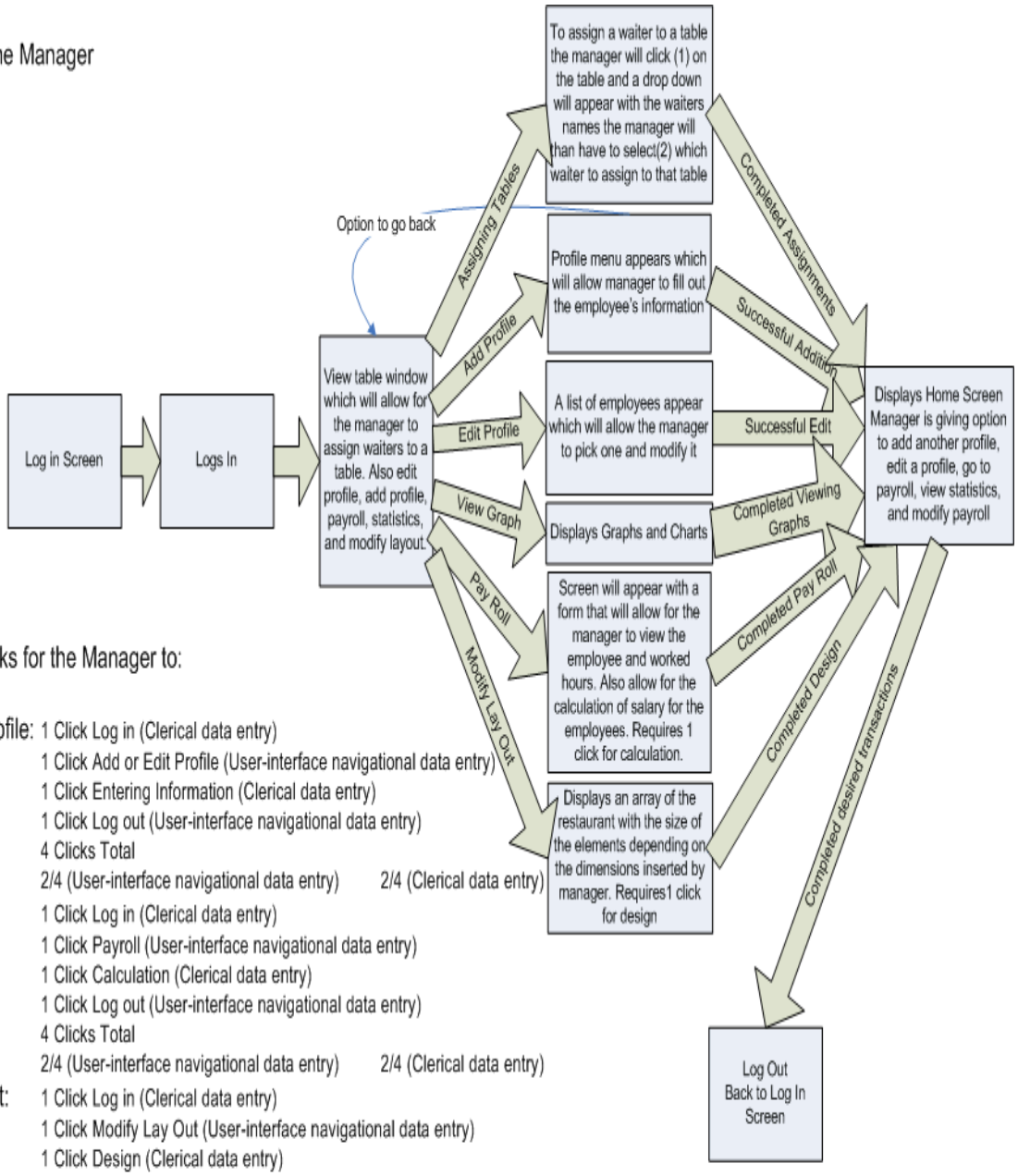
Total amount of clicks for the waiter to:

-Check status of Order:
1 Click Log in (Clerical data entry)
1 Click Log out (User-interface navigational data entry)
2 Clicks Total
1/2 (User-interface navigational data entry) 1/2 (Clerical data entry)

-Add Item to Table:
1 Click Log in (Clerical data entry)
1 Click Add Item (User-interface navigational data entry)
3 Clicks Choosing Desired Item (User-interface navigational data entry)
1 Click Add Item to Tab (User-interface navigational data entry)
1 Click Log out (User-interface navigational data entry)
7 Clicks Total
6/7 (User-interface navigational data entry) 1/7 (Clerical data entry)

Manager

Possible Paths for the Manager



Total amount of clicks for the Manager to:

-Add or Edit Profile: 1 Click Log in (Clerical data entry)
1 Click Add or Edit Profile (User-interface navigational data entry)
1 Click Entering Information (Clerical data entry)
1 Click Log out (User-interface navigational data entry)
4 Clicks Total
2/4 (User-interface navigational data entry) 2/4 (Clerical data entry)

-Pay Roll: 1 Click Log in (Clerical data entry)
1 Click Payroll (User-interface navigational data entry)
1 Click Calculation (Clerical data entry)
1 Click Log out (User-interface navigational data entry)
4 Clicks Total
2/4 (User-interface navigational data entry) 2/4 (Clerical data entry)

-Modify Lay Out: 1 Click Log in (Clerical data entry)
1 Click Modify Lay Out (User-interface navigational data entry)
1 Click Design (Clerical data entry)
1 Click Log out (User-interface navigational data entry)
4 Clicks Total
2/4 (User-interface navigational data entry) 2/4 (Clerical data entry)

-Assigning Tables: 1 Click Log in (Clerical data entry)
1 Click Table (User-interface navigational data entry)
1 Click Name from Drop-Down Menu (User-interface navigational data entry)
1 Click Log out (User-interface navigational data entry)
4 Clicks Total
3/4 (User-interface navigational data entry) 1/4 (Clerical data entry)

Description of Modifications and Rationale

Login Screen:

Description of Enter Button:

This will allow for the entered data to be verified and if verified directed to the appropriate next screen depending on the employee's type.

Host Screen:

Description of Assign Button:

This will change the color of the selected table to YELLOW signifying that the current table is now occupied. It will also update the waiter that has been assigned to the table that it is now awaiting service.

Description of Queue Button: (Modified)

This will allow for tracking of how many customers are currently waiting to be seated. There will also be a text box displaying how many groups are currently waiting in the queue list. The number of groups in the queue list will automatically be deducted when the host assigns a group to a table and if there are no groups currently in the queue list it will leave the group number at zero.

Reason for modification:

This feature will allow for a real time update of the queue list. This queue list will allow for the host to have on-hand information about the current wait times should the restaurant become very busy.

Description of the Logout Button:

This will allow for the Host to log out of the current system.

Bus Boy Screen:

Description of Ready Button:

This will change the status of the selected table(s) to GREEN and update the Host's Screen.

Description of Logout Button:

This will allow for the Bus Boy to log out of the current system.

Cook Screen:

Description of Ready Button: (Modified)

This will check to confirm that the orders selected are valid. If so they will be transferred to the bottom portion of the screen, which signifies that they are currently being completed.

Reason for modification:

This was done to make sure that the appropriate order combination was completed. This was implemented so an order for a dessert would not arrive before the order for the main entrée. This would allow for a more realistic approach to the order.

Description of the Completed Button: (Modified)

This will notify the appropriate waiter that the selected order has been completed and ready to be taken. It will than remove that order from the screen completely.

Reason for modification:

This would allow for the easy implementation of the above process.

Description of the Logout Button:

This will log out the cook from the current system.

Manager Screen: (Main Screen)

Description of Edit Profile Button:

This will lead the manager to a new screen (Manager Search Screen) that will give the manager the fields that he/she would like to search to find the desired profile. The new screen will have a blank profile screen

Description of Add Profile Button:

This will lead the manager to a new screen (Manager Add Profile Screen) that will contain a blank profile, which the manager will be able to fill out with the information of the new employee he is adding to the system.

Description of Payroll Button:

This will lead the manager to a new screen (Manager Payroll Screen) that will contain a list of all the employees with the following fields: date of employment, type of employee, name, and address.

Description of Statistics Button:

This will lead the manager to a new screen (Manager Statistics Screen) that will contain descriptive graphs pertaining to the inventory, the highest selling item, the amount of customers during the day, week or monthly and the salary of all the employees.

Description of Assign Button:

This will allow for the manager to assign the selected table with the desired waiter and busboy. Assigning a waiter to a table will be done by selected a drop down menu above the desired table and choosing which waiter shall service the current table. There will also be another drop down window for the manager to choose which bus boy will be serving the current table.

Description of Modify Button:

This will lead to a new screen (Manager Modify Layout Screen) that will allow the manager to enter the dimensions of the table he would like to use for the restaurant.

Description of Logout Button:

This will log out the manager from the current system.

Manager Add Profile Screen:

Description of Add Profile Button:

This will check to make sure all the appropriate fields in the Profile have been filled out. Once proved valid the information will be stored and the system will be updated.

Description of Go Back Button:

This will allow the manager to go to the previous Manager Main Screen. This has been implemented incase the manager mistakenly pressed the wrong button.

Description of Logout Button:

This will log out the manager from the current system.

Manager Search Screen:

Description of Search:

This will lead to a new screen (Manager Result Screen) that will contain all the matching information given in a list. It will make sure that something was actually entered into a search field.

Description of Go Back:

This will allow for the manager to go back to the Main Manager Screen.

Description of Logout:

This will log out the manager from the current system.

Reason for modification:

This new screen was added to assist the manager in the editing process of the desired profile. This screen allows for the manager to modify his/her search as to create a more manageable list of employee names. This would help further refine his search for the correct employee.

Manager Result Screen:

Description of Edit Button:

This will lead to a new screen (Manager Edit Profile Screen) that will contain the information of the desired employee selected.

Description of Go Back Button:

This will allow for the manger to go back to the Main Manager Screen.

Description of Logout Button:

This will log out the manager from the current system.

Reason for modification:

This new screen was created to give the manager a more refined list of employees for a quick glance to allow him/her to find the desired employee quickly.

Manager Edit Profile Screen:

Description of Edit Button:

This will save the information that was changed to the current profile and store it for later use. It will replace the information that was previously stored for that employee.

Description of Go Back Button:

This will allow for the manager to go back to the Main Manager Screen.

Description of Logout Button:

This will log out the manager from the current system.

Manager Payroll Screen:

Description of Filter Button:

This will sort the employee's list depending on the selected fields.

Description of Go Back Button:

This will allow for the manager to go back to the Main Manager Screen.

Description of Logout Button:

This will log out the manager from the current system.

Manager Statistics Screen:

Description of Go Back Button:

This will allow for the manager to go back to the Main Manager Screen.

Description of Logout Button:

This will log out the manager from the current system.

Manager Modify Layout Screen:

Description of Design Button:

This will calculate the size of the elements in the array by using the specified dimensions entered. It will show a layout of the restaurant split up into an array which will be able to accommodate the size of the tables specified.

Description of Go Back Button:

This will allow for the manager to go back to the Main Manager Screen.

Description of Logout Button:

This will log out the manager from the current system.

Waiter Main Screen:

Description of Table Button:

This will lead to a new screen (Waiter Tab Screen) which will allow the waiter to see the current tab on the table and will be given the option to add another item to the tab or calculate and close the table.

Description of Logout Button:

This will log out the waiter from the current system.

Waiter Tab Screen:

Description of Add Item Button:

This will lead to a new screen (Waiter Add Item Screen) which will allow the waiter to add additional items to the current tab for that table.

Description of Close Tab Button:

This will close the current tab on the table sending a message to print the receipt for the table and than it will change the status of the table to dirty which is signified by the color YELLOW.

Description of Go Back Button:

This will allow for the waiter to go back to the Waiter Main Screen.

Description of Logout Button:

This will log out the waiter from the current system.

Waiter Add Item Screen:

Description of Add Item Button:

This will allow for the current item selected to be added onto the current tab of the table that is currently selected. The waiter will select an item by specifying whether it is an entrée, desert, or appetizer. The waiter will than select the food category. Afterwards the waiter will select the desired meal. It is done in this order to help specify the meals priority. For instance if a table orders their entrée and dessert at the same time, the system will know to send the order for the entrée to the cook first and keep the dessert on queue for a predetermined amount of time. After that time has passed it will than send the cook the order for the dessert.

Description of Go Back Button:

This will allow for the waiter to go back to the Waiter Main Screen.

Description of Logout Button:

This will log out the waiter from the current system.

History of Work and Current Status of Implementation

History of Work

History of work is attached to the back of the report.

Current Status of Implementation

Everything is generally working; the team is working on resolving a number of bugs in various parts of the system. We are well on track to finish everything we have set out to finish by the due date.

Conclusions and Future Work

As one would expect on a project of any significant magnitude, we did in fact encounter a number of technical challenges on our way to successfully meeting our design objectives. These challenges included such issues as communication, environmental, and architectural issues. Only though the utilization of the software engineering approaches and techniques taught in the class, as well as through the indispensable assistance of the faculty and staff, were we ably to successfully tackle these various challenges.

The very first hurdle that we had to clear as a team was setting up an effective development environment that would align well with the development technologies we have chosen. After a fare amount of discussion and elaboration, we have decided that the best way to approach development in PHP and MySQL was to use a simple text editor designed for experienced coders, such as MultiEdit, PSPad, or any other similar shareware product available free of charge on the Internet. Due to the choice of PHP as our programming language of choice, we have also decided to use Microsoft Visio for UML purposes, since the bulk of the UML automation tools are designed to work with Sun Java. These choices were both influenced and aided by the excellent capabilities of the university computing labs made available to us, and the intrinsic abilities of the webs hosting services that we have chosen. Prior web development experience on the part of several key members of the team has also greatly contributed to the excellent choices made to resolve issues of this sort.

Upon the setup of an efficient development environment, we have encountered several other issues that we have gone to successfully resolve using various software

engineering techniques that we have learned in the class. A persistent and difficult issue was communication between the team members as far as the interaction amongst the components that they individually created. This issue was again and again resolved by referring to the UML documentation which acted as the final authority in every dispute that has arisen during the development process. It is very clear to us now that without the proper documentation, UML or any other, even a project which evolves as few as six people would quickly get stuck in the quicksand of component incongruence. One aspect of prior knowledge which has aided the team greatly in resolving these unavoidable communication conflicts was the conflict resolution and management experience of our team leader.

Another technical challenge we have run into along the course of this project was the choice for the overall architecture of the project. In resolving this issue we were greatly aided by the material covered in the class regarding pattern reuse and other similar topics. Prior development experience has also factored in greatly into resolving this very important issue. After much deliberation amongst the entire group, during which every member of the team had an opportunity to contribute to the process, we had made a decision to build around the Model/View/Controller architecture. In our opinion this format offered great flexibly that we felt was absolutely necessary for our product because a restaurant automation system would have to be greatly customizable in order to have a chance of being profitable in the modern marketplace. The great degree of decoupling between interface, data, and control procedures would allow us to write flexible and extensible code that should in at least in theory have the capability to match the business logic demands of a diverse group of customers.

We have also encountered a number of smaller, coding level challenges as we labored through the implementation of the various business logic objectives we have setup for ourselves. One such issue was the decision on how to maintain the ongoing and evolving relationship between a waiter, the orders he has submitted to the system, the table these orders came from, the cook who takes and cooks these orders, and the logging needed to document this process for managerial purposes. Initially we were going to implement this relationship without pulling them together in any centralized way by having the various objects involved communicate asynchronously amongst each other. This approach however posed some serous challenges to the goals we have set out to accomplish. The biggest problem with that approach and the problem that ultimately lead to its demise was the fact that logging the necessary information without coming with a pulled representation for all the data necessary was virtually impossible. Faced with this issue we went on to modify out initial approach and came up with the concept of the session, both a runtime and a database object which pulls all the information about the aforementioned interobject interactions together into one entity. We were greatly influenced by and aided by the UML documentation process as without it we would have realized the serious problems our initial approach would cause way to late in the development process. We were specifically greatly aided by the class diagram and interaction diagram parts of the UML documentation. As far as prior experienced is concerned, several member of our group have worked with relational databases in the past and used this valuable experience to tackle the issue once it has been discovered.

An additional technical challenge which we have faced is the following. As one of our two points of emphasis for our design, we had set out to implement a floor layout

manager which would allow the manager to create a layout of the restaurant for the overall systems to use. This was an interesting design challenge into which we have put a lot of thought. The modular development approach that has been taught to us in class has come in handy to solve this problem as well. Our solution amounted to essentially dividing the restaurant floor into a two dimensional coordinate grid, with adjustable magnitude of the scale. The smaller the scale magnitude for the coordinates, the more curvature present in the design, and vice versa. The size of the image representations of tables would also vary accordingly. The manager is expected to enter the grid coordinates for the tables in order to create a layout. As a possible upgrade feature that could be implemented in the future it is possible to create a more intuitive user friendly layout creation tool. It is also possible to save various layouts and allow the manager to choose from these previously created layouts for his design. The mathematical background which all of the members of our team possess as a result of participating in the general engineering curriculum has contributed greatly to resolving this particular issue.

We have faced a lot of technical challenges, some mentioned here, and some not, but the training and support we have received from the faculty and stuff have allowed us to overcome them all. UML documentation has resolved conflicts, guided the overall development process, and kept us concourse when nothing else could. Should we ever choose to extend this project in the future, there are several directions in which we can go. We can add more functionality to the managerial statistics interface; improve on the layout manager, the menu manager, and the general look and feel of the entire system. We could also actually test out the system under at least semi real conditions for feed back. Overall this has been an a very fulfilling and challenging experience.

Breakdown of Responsibilities

Simon Rudin – Project Manager

- Class Diagram and Interface Specification (Major Role)
- System Architecture and System Design (Minor Role)
- Logistics
- Database Design (Major Role)
- Progress Report and Plan of Work
- Integration Coordinator

Development Assignments:

- Everybody has contributed equally to the coding part of the project.

David Elrom – Lead Developer

- Class Diagram and Interface Specification (Minor Role)
- System Architecture and System Design (Major Role)
- References
- Database Design (Major Role)
- Website Creation and Maintenance
- PHP Expert

Development Assignments:

- Everybody has contributed equally to the coding part of the project.

James Rosado – System Analyst

- Interaction Diagrams
- System Architecture and System Design (Minor Role)
- Database Design (Minor Role)
- Visio Expert
- Meeting Notes Taker
- Logistics (Minor Role)

Development Assignments:

- Everybody has contributed equally to the coding part of the project.

Christian Santiago – Developer

- User Interface Design and Implementation
- System Architecture and System Design (Minor Role)
- Interaction Diagrams (Minor Role)
- Documentation Expert
- Database Design (Minor Role)
- Integrated Systems Tester

Development Assignments:

- Everybody has contributed equally to the coding part of the project.

Summary of Changes

- Menu Manage use case added.
- Logout use case added.
- Changed Report Dirty Table to Close Tab, which both changes table status to dirty and closes the table tab.
- Check Identity use case changed to Login use case.

References

- **Gehrke, Johannes. Ramakrishnan, Raghu. Database Management Systems, Third Edition.**
- **Bruegge, Bernd. Dutoit, Allen. Object-Oriented Software Engineering, Second Edition.**
- **Miles, Russ. Hamilton, Kim. Learning UML 2.0 First Edition**
- **PHP Official Documentation**

  http://www.php.net/manual/en/
- **MySQL Official Documentation**

  http://dev.mysql.com/doc/
- **Dev Shed**

  http://www.devshed.com
- **Zend Developer Resources**

  http://devzone.zend.com