Report 2: SYSTEM DESIGN

Group 11

14:332:452

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

Jazmin Garcia
Eric Gilbert
Greg Paton
Vishal Shah
Damon Chow

Table of Contents

I. Business Policies

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

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

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

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

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

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

II. Interaction Diagrams

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

i. CheckTablesStatus

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

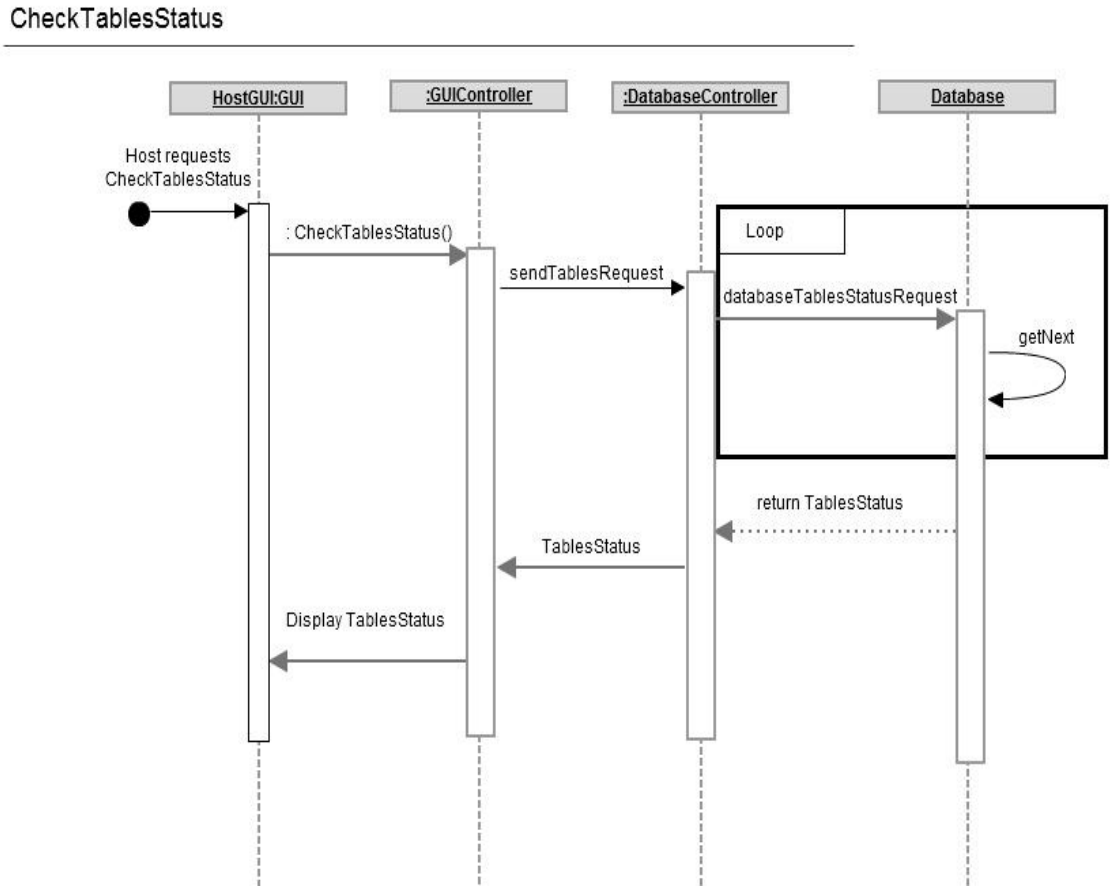

Figure 2.1: CheckTableStatus interaction diagram.

ii. PlaceOrder

This interaction exhibits high cohesion as no object has many responsibilities. The Controller and has other functionalities when dealing in other use-cases and as a result it is not tasked with the responsibility of the specialized object OrderArchiver or Order. The coupling in this interaction is not low with respect to controller. It has the responsibility of communicating with each other object. DatabaseConnection exhibits expert doer principle because it retrieves data from the database; which is a system known only to this object.
The interaction for placing an order proceeds as followed:
User enter and order on the UserInterface which is then requested to be added through AddOrderRequest, which creates an Order. Then the order is sent to the controller, which verifies with that database that the order can be placed. At this point there are two scenarios. The first scenario is that the order is valid and then a loop is run until the order can be saved by the OrderArchiver. Finally, the Controller tells the UserInterface to prompt that the order was placed. The second scenario is when the order is not valid. When this occurs the Controller tells the UserInterface to prompt that the order was not placed.

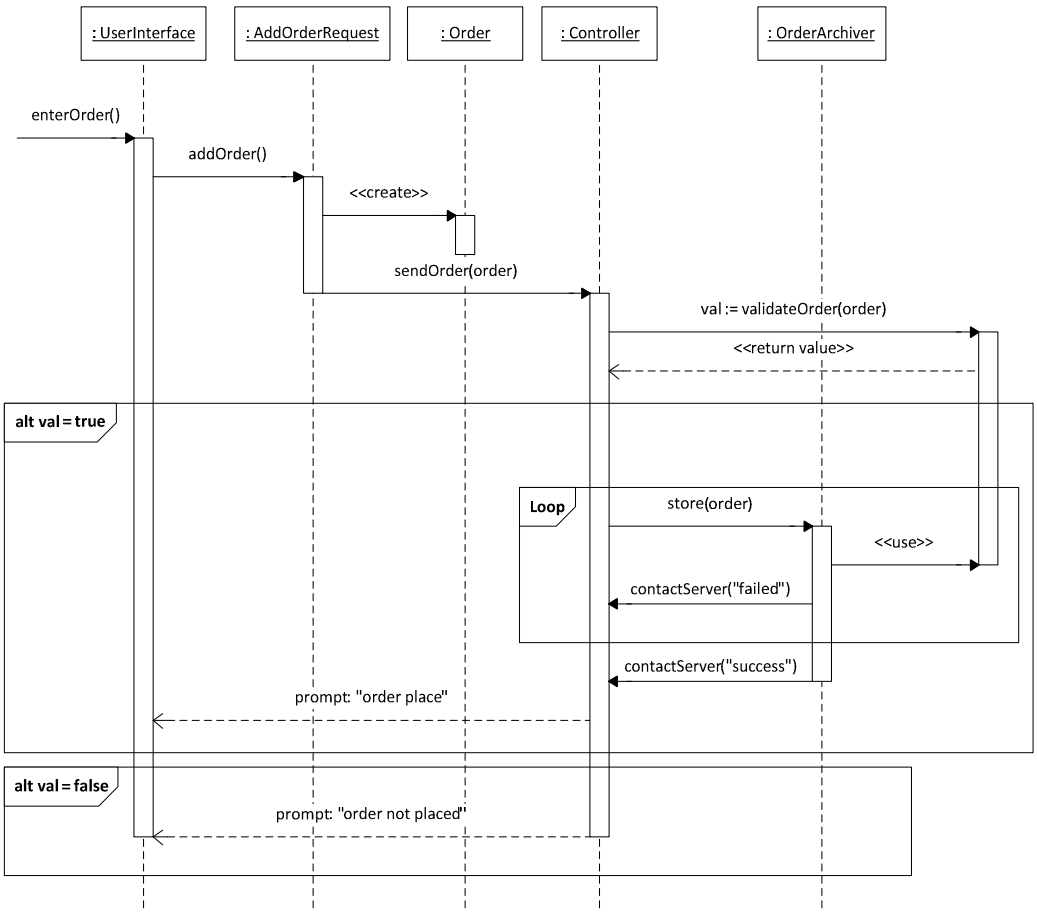

Figure 2.2: PlaceOrder interaction diagram.

iii. UpdateEmployee

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

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

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

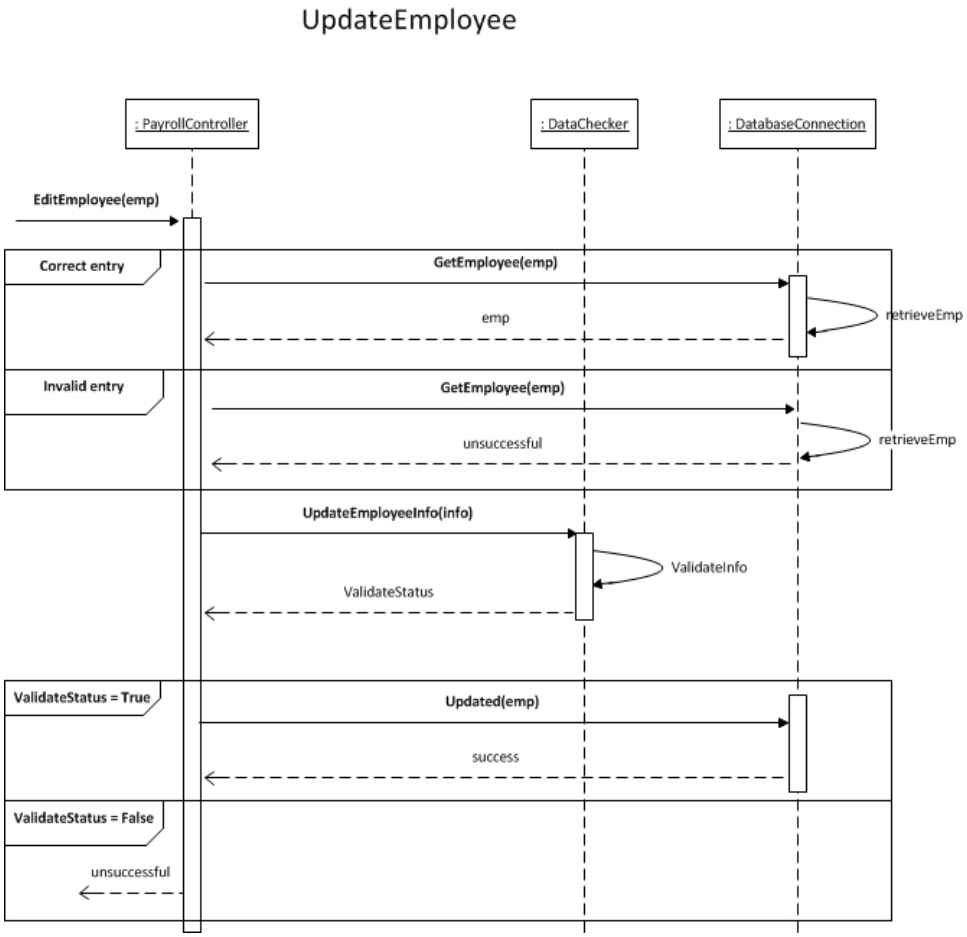

Figure 2.3: UpdateEmployee interaction diagram.

iv. EditMenu

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

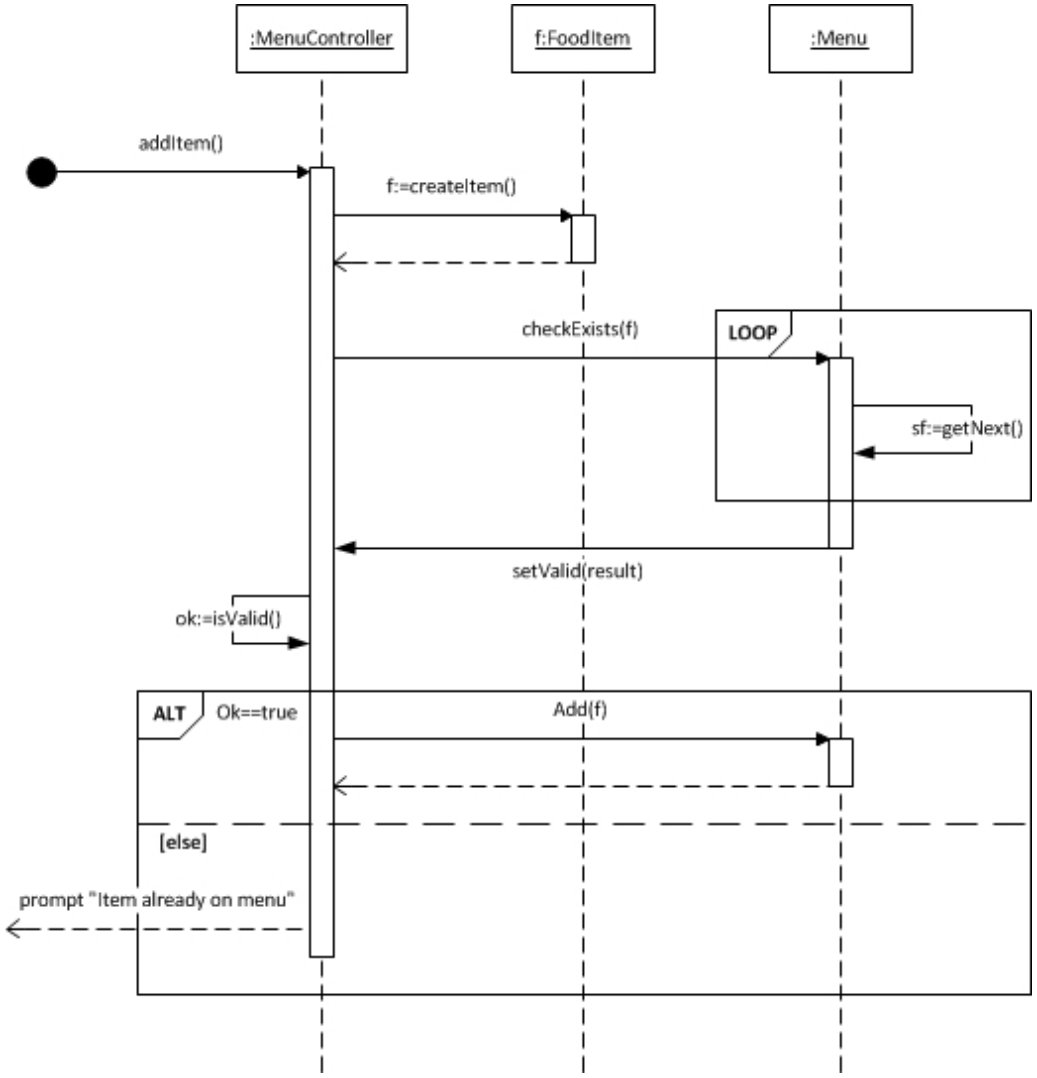

Figure 2.4a: AddItem interaction diagram.

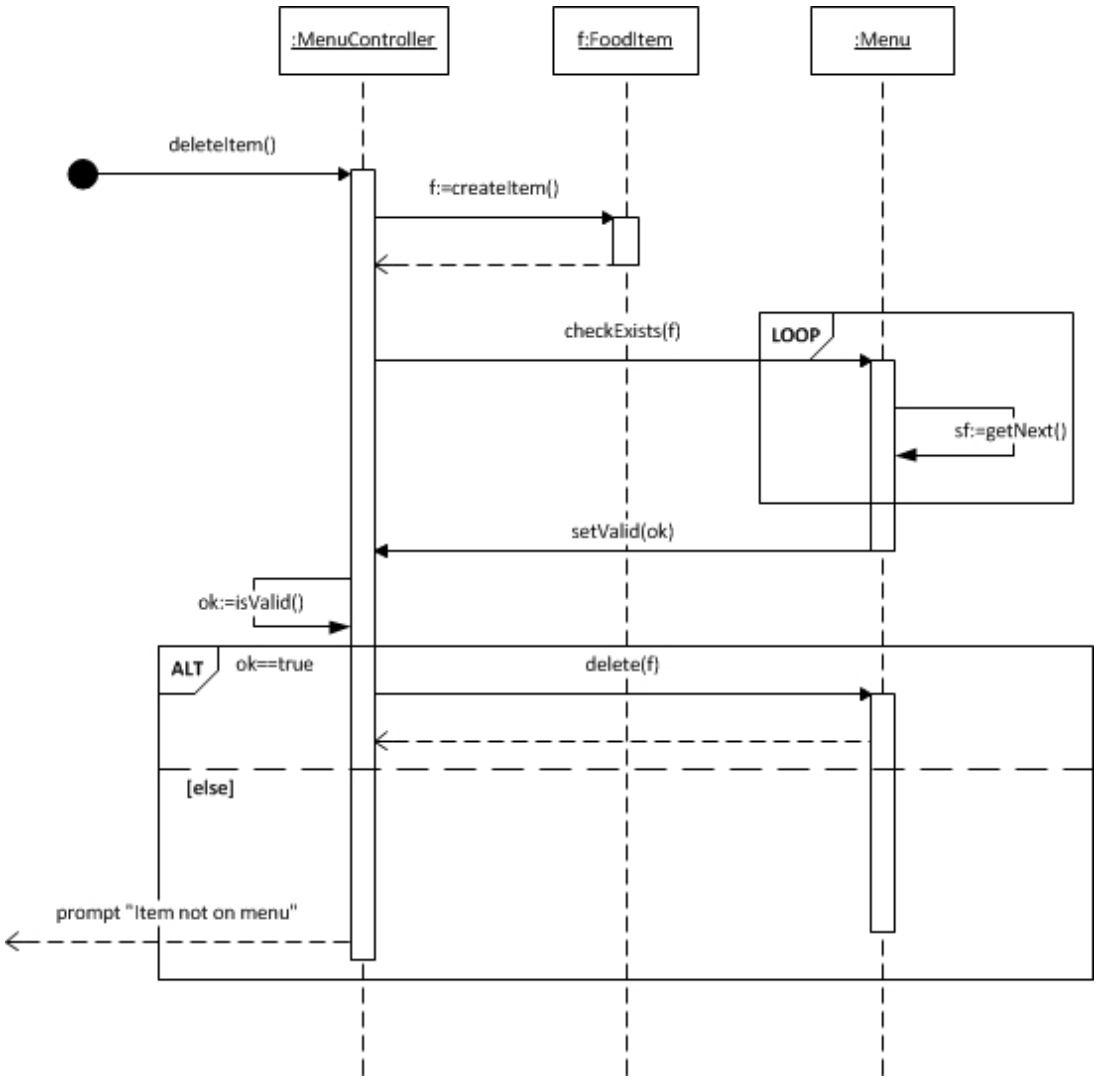

Figure 2.4b: DeleteItem interaction diagram.

iv. CheckStats

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

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

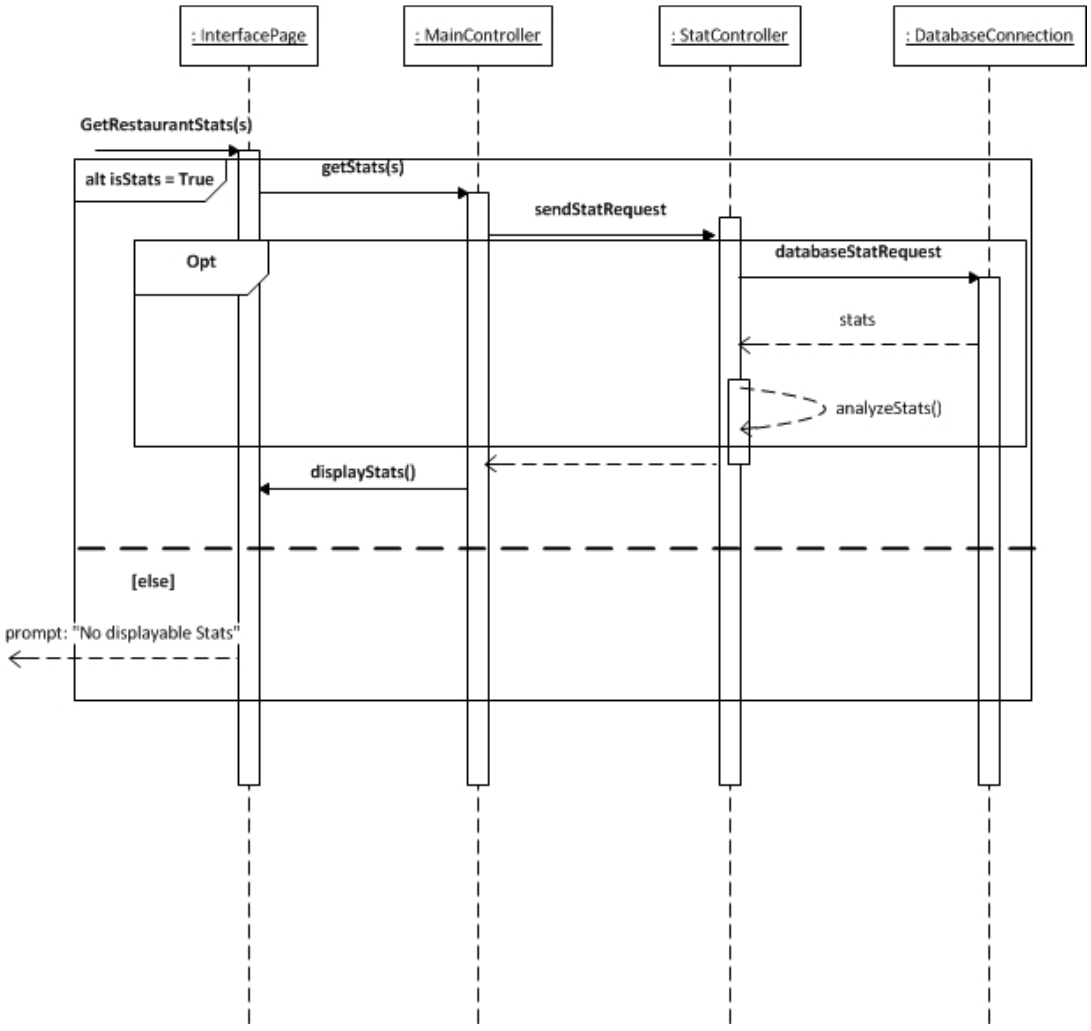

Figure 2.5: CheckStats interaction diagram.

III. Class Diagram and Interface Specification

a. Class Diagram

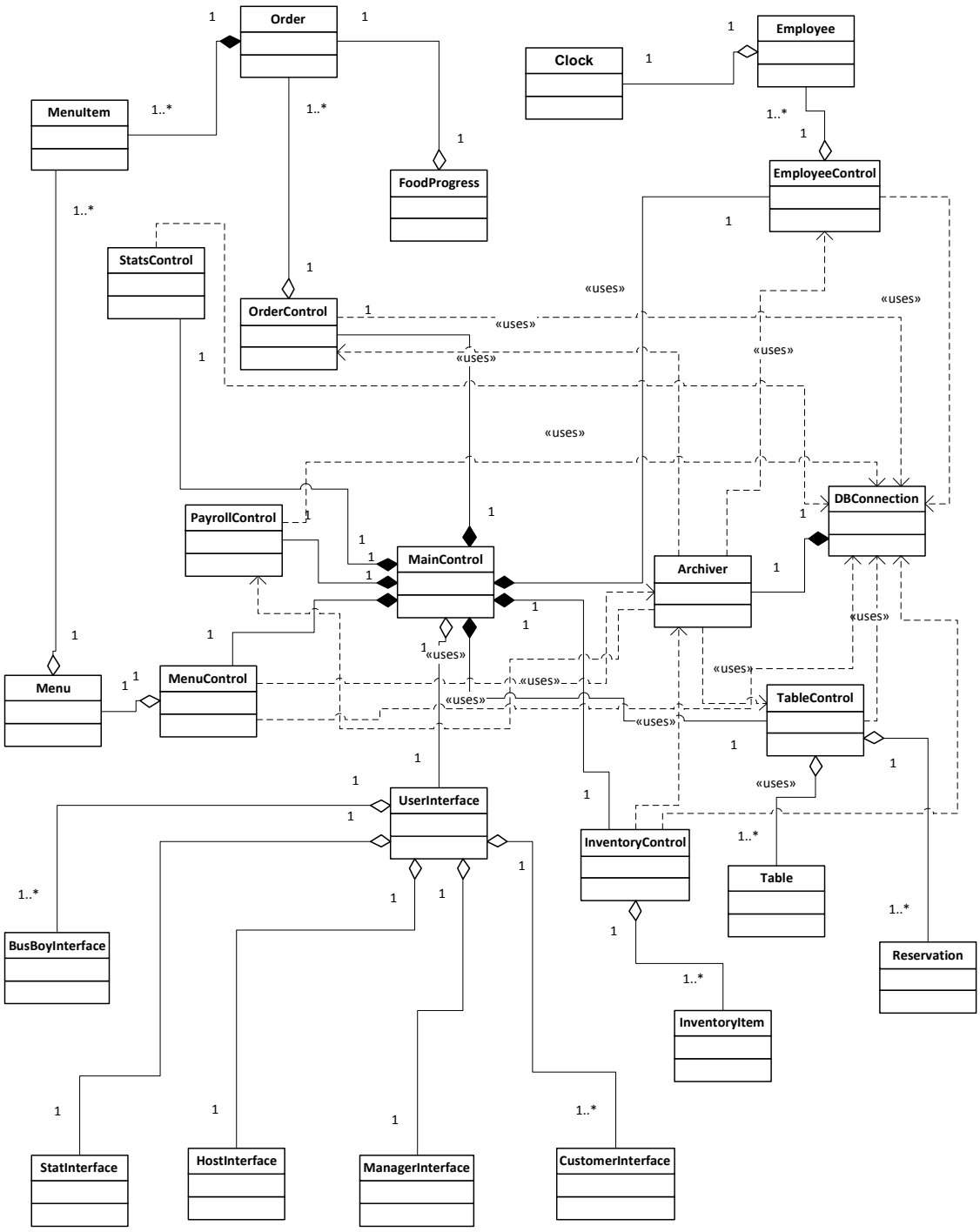

Figure 2.6: An overview of the entire class diagram.

i. Detailed Class Diagram Part 1

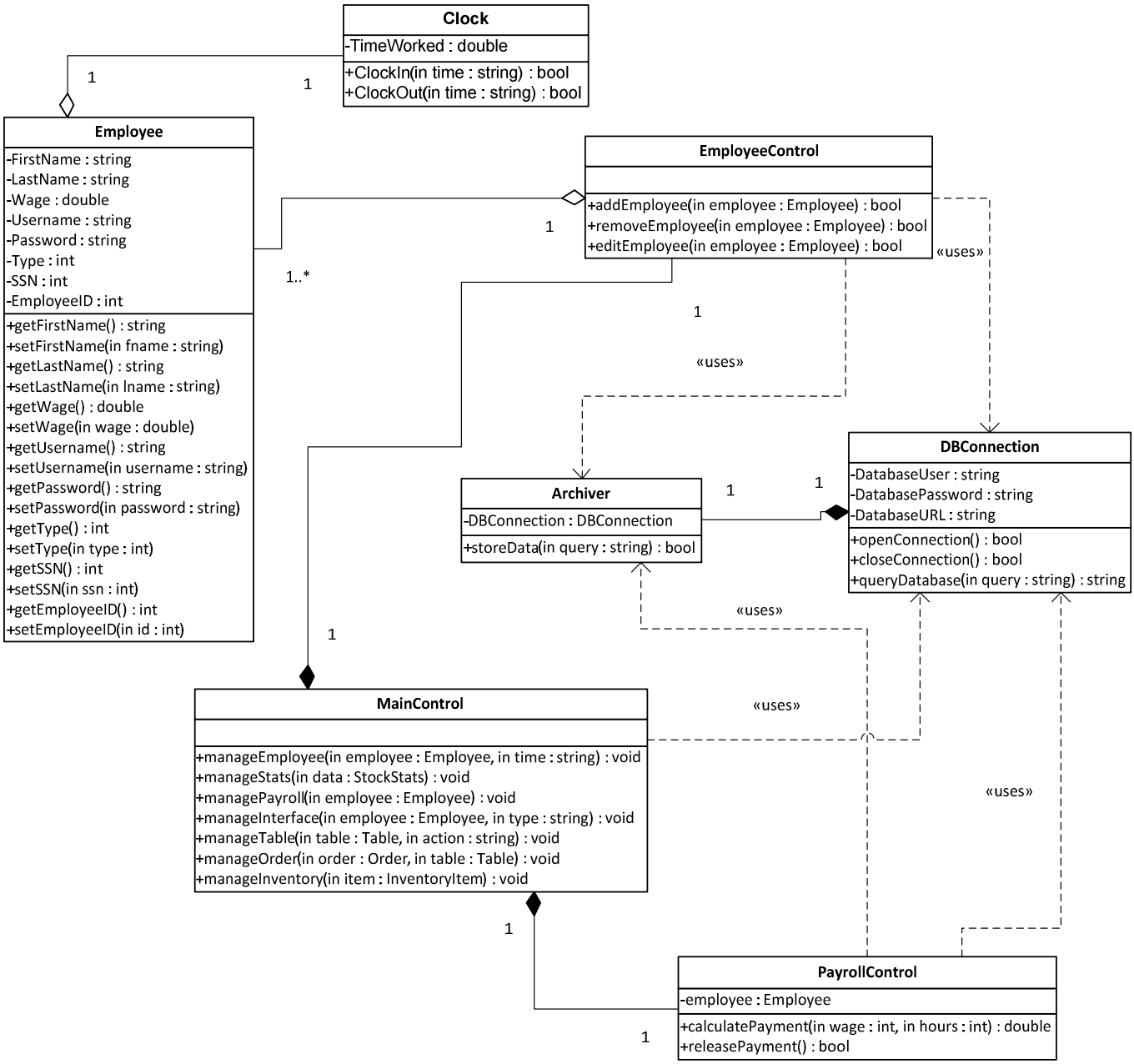

Figure 2.7a: Employee and Payroll control part of the class diagram.

ii. Detailed Class Diagram Part 2

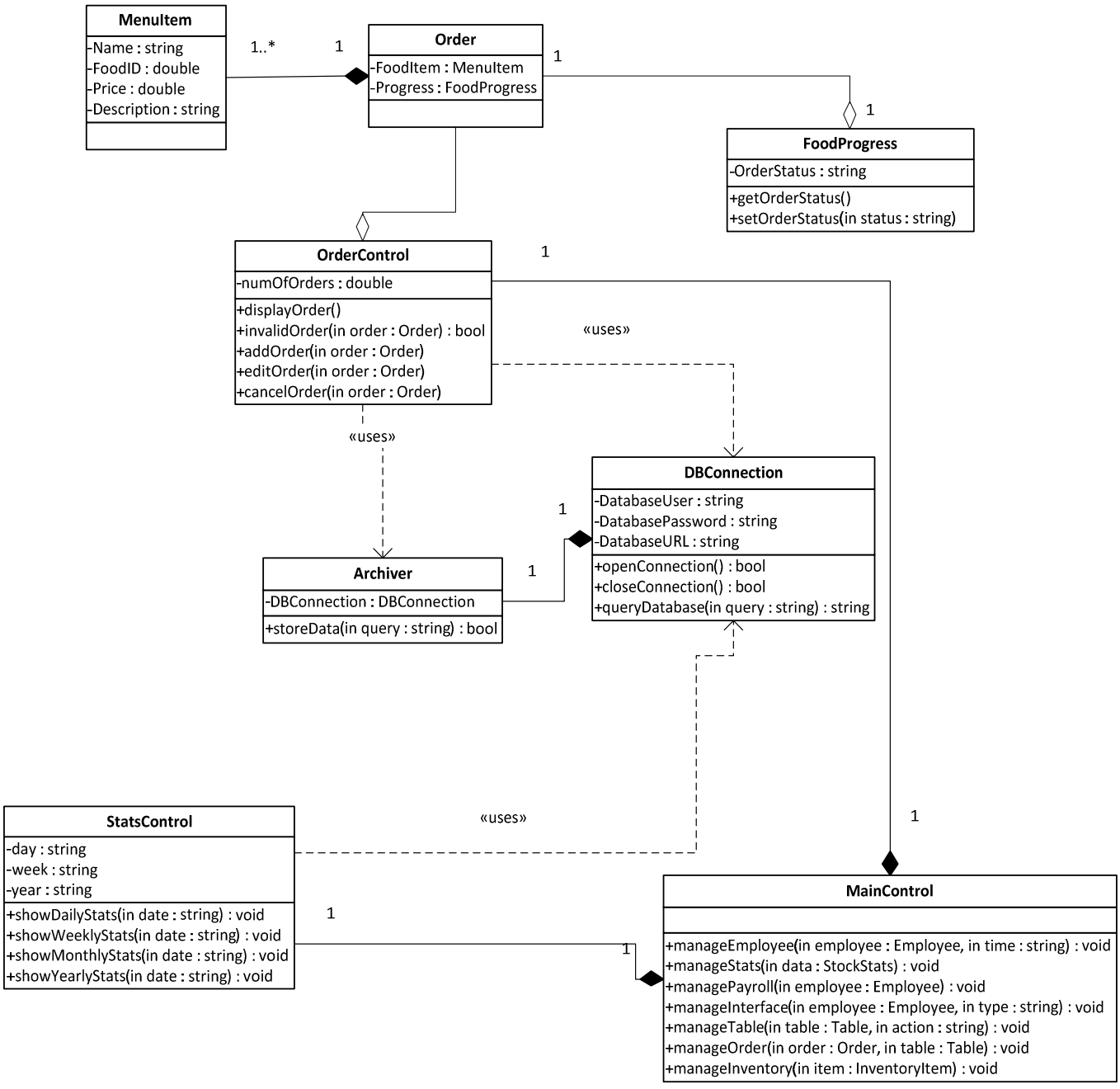

Figure 2.7b: The StatsControl and OrderControl parts of the class diagram.

iii. Detailed Class Diagram Part 3

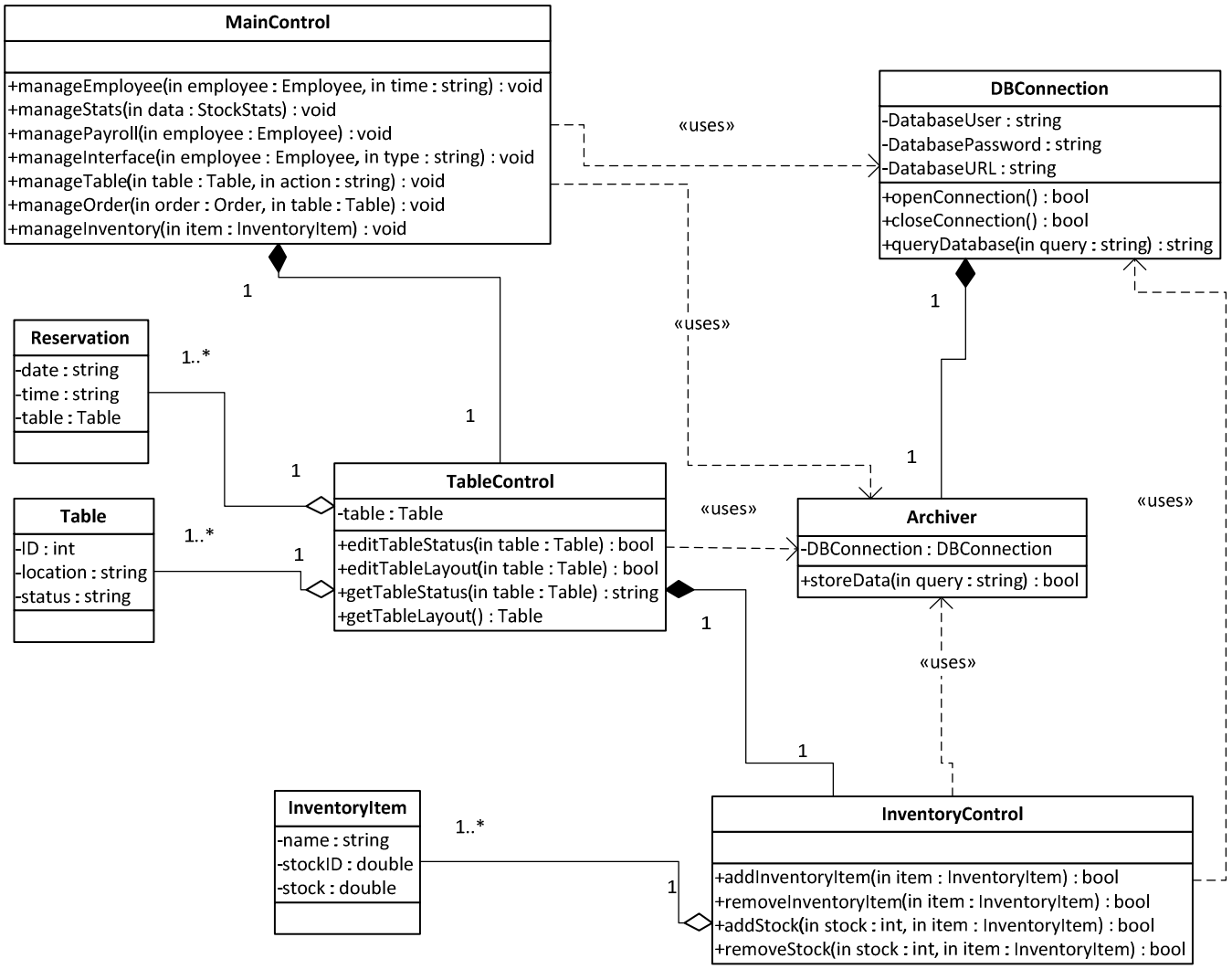

Figure 7c: The TableControl and Inventory Control parts of the class diagram.

iv. Detailed Class Diagram Part 4

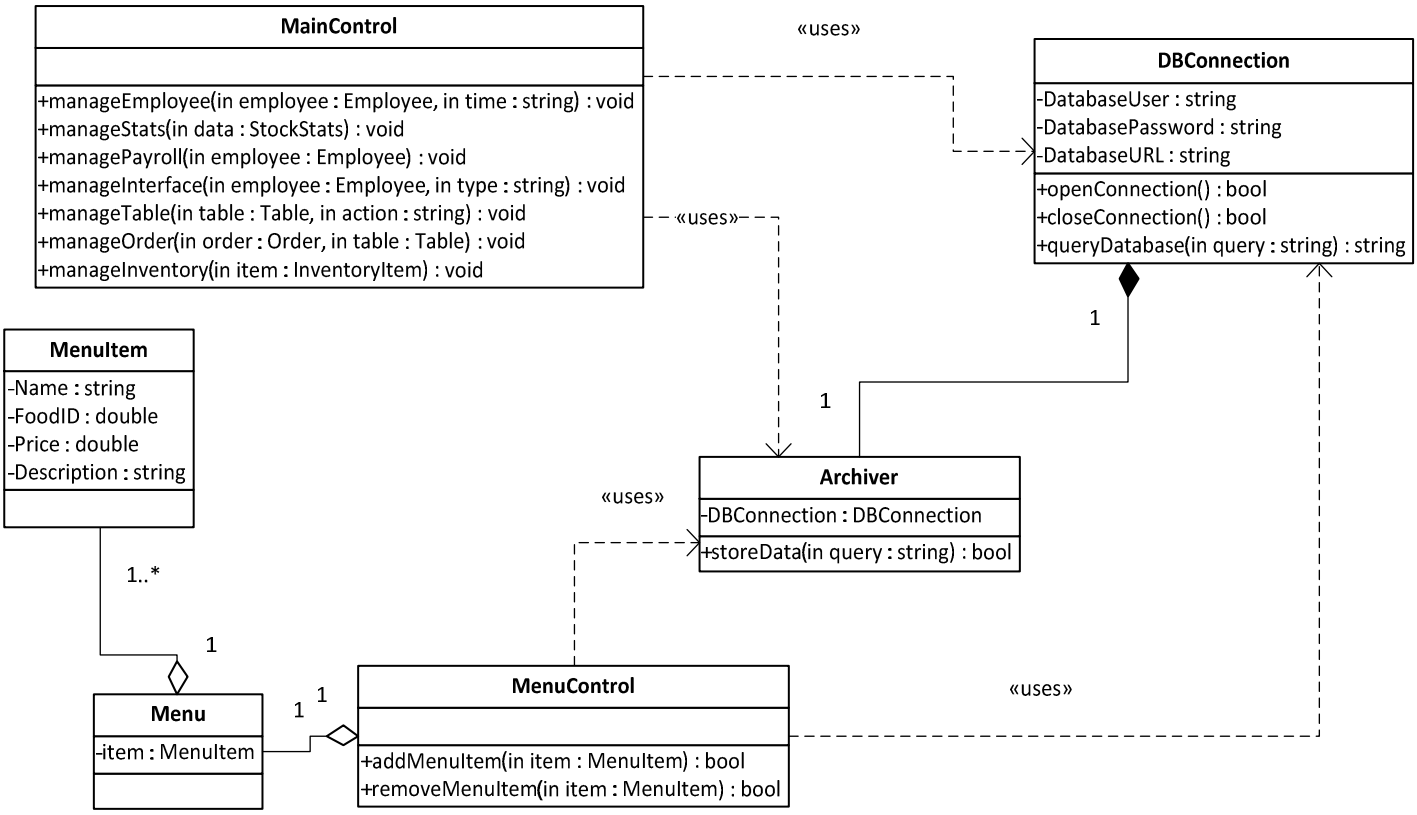

Figure 2.7d: The MenuControl part of the class diagram.

v. Detailed Class Diagram Part 5

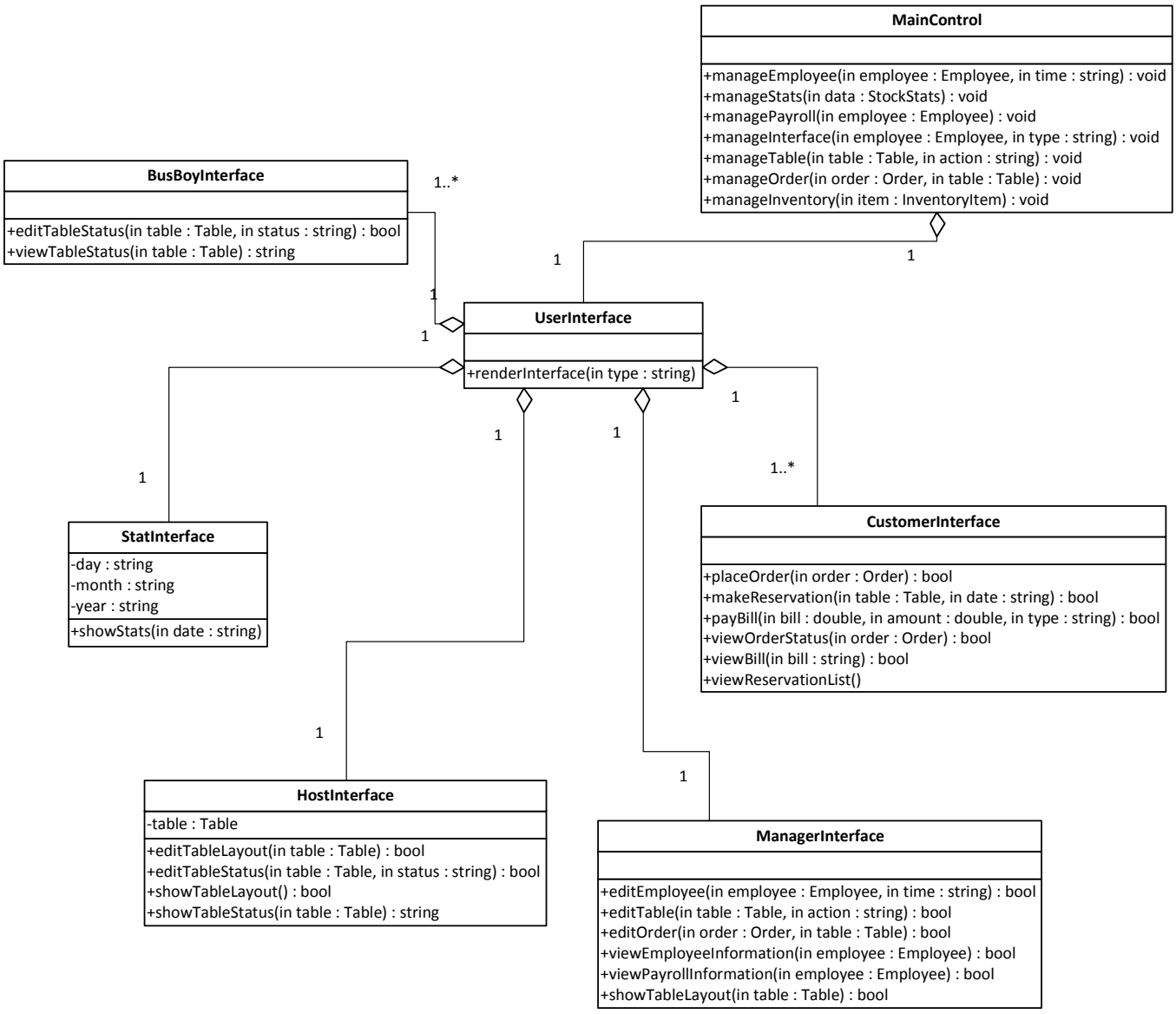

Figure 2.7e: All the user interface classes that are part of the entire class diagram.

b. Data Types and Operation Signatures

MainController

Attributes:

Operations:

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

DBConnection

Attributes:

-DatabaseUser:string
//The username for all employee logins
-DatabasePassword:string
//The password for all employee logins
-DatabaseURL:string
//The location of the database

Operations:

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

Archiver

Attributes:

-DBConnection:DBConnection

//The connection with database

Operations:

+storeData(in query:string) : bool

//Stores data from query to the database

PayrollControl

Attributes:

-employee:Employee

//The employee referred to for payroll

Operations:

+calculatePayment(in wage:int, in hours:int) : double

//Determines the amount earned for the employee

+releasePayment() : bool

//Pays the employee the amount earned and resets hours worked

Clock

Attributes:

-TimeWorked : double

// amount of time an employee has worked

Operations:

+ClockIn(in time:string) : bool

// records the time when an employee starts working

+ClockOut(in time:stirng) : bool

//records the ending time of an employee's work shift

EmployeeControl

Attribute:

Operations:

+addEmployee(in employee:Employee) : bool

//Puts a new employee profile into the database

+removeEmployee(in employee:Employee) : bool

//Deletes and existing employee profile from the database

+editEmployee(in employee:Employee) : bool

//Changes the information in the employee profile in the database

Employee

Attributes:

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

Operations:

+getFirstName() : string
//Returns the employee's first name
+setFirstName(in fname:string)
//Changes/makes first name to fname
+getLastName() : string
//Returns the employee's last name
+setLastName(in lname:string)
//Changes/makes last name lname
+getWage() : double
//returns the employee's wage
+setWage(in wage:double)
//Changes/makes the wage as specified
+getUsername() : string
//Returns the employee's username
+setUsername(in username:string)
//Changes/makes the username as specified
+getPassword() : string
//Returns the employee's password
+setPassword(in password:string)
//Changes/makes the password as specified
+getType() : int
//Returns the employee's occupational role
+setType(in type:int)
//Changes/makes the employee's occupational role as specified
+getSSN() : int
//Returns the social security number of an employee

+setSSN(in ssn:int)
//Changes/makes the social security number a 9 digit integer
+getEmployeeID() : int
//Returns the employee's identification number
+setEmployeeID(in id:int)
//Changes/makes the employee's identification number as specified.

MenuItem

Attributes:

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

Operations:

Order

Attributes:

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

Operations:

OrderControl

Attributes:

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

Operations:

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

FoodProgress

Attributes:

-OrderStatus : string

//variable that stores whether an order is started, being prepared, or finished

Operations:

+getOrderStatus()

//Returns the whether an order is started, being prepared, or finished

+setOrderStatus(in status:string)

//Makes an order with the status specified

StatsControl

Attributes:

-day : string

//The day

-week : string

//The week

-year : string

//The year

Operations:

+showDailyStats(in date:string) : void

//displays the financial statistics for every day starting with the inputted date

+showWeeklyStats(in date:string) : void

//displays the financial statistics by weeks starting with the inputted date

+showMonthlyStats(in date:string) : void

//displays the financial statistics by months starting with the inputted date

+showYearlyStats(in date:string) : void

//displays the financial statistics by year starting with the inputted date

InventoryItem

Attributes:

-Name : string

//name of food item in inventory

-StockID : double

//identification number for food item in inventory

-Quantity : double

//amount of the food item in the inventory

Operations:

InventoryControl
Attributes:

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

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

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

TableControl
Attributes:
-table:Table
//A table in the restaurant
Operations:
+editTableStatus(in table:Table) : bool
//Changes the status of a table
+editTableLayout(in table:Table) : bool
//Moves a table to a different location
+getTableStatus(in table:Table) : string
//Returns the status of a table
+getTableLayout() : Table
//Returns the layout of the tables

Menu

Attributes:

-item:MenuItem

//An item on the menu

Operations:

Menu Control

Attributes:

Operations:

+addMenuItem(in item:MenuItem) : bool

//Adds an item to the menu

+removeMenuItem(in item:MenuItem) : bool

//Removes an item from the menu

BusBoyInterface

Attributes:

Operations:

+editTableStatus(in table:Table, in status:string) : bool

//Changes the table to clean

+viewTableStatus(in table:Table) : string

//Returns the status of a table (dirty, reserved, occupied, clean)

CustomerInterface

Attributes:

Operations:

+placeOrder(in order:Order)

//Make an order from the menu and submit it to the chef.

+makeReservation(in table:Table, in date:string)

//Reserve a table(s) in the restaurant for a specified date.

+payBill(in bill:double, in amount:double, in type:string)

//Pays the final bill with either cash or credit card.

+viewOrderStatus(in order:Order) : bool

//Shows the progress of an order

+viewBill(in bill:string) : bool

//Shows the final bill

+viewReservationList()

//Shows all the reservations made

StatInterface

Attributes:

-day : string

//The day

-month : string

//The month

-year : string

//The year

Operations:

+showStats(in date:string)

//Displays the financial and customer statistics for the inputted date.

HostInterface

Attributes:

-table : Table

//Represents an actual table in the restaurant.

Operations:

+editTableLayout(in table:Table) : bool

//Moves the selected table to a specified new location.

+editTableStatus(in table:Table, in status:string) : bool

//Denotes the selected table as dirty, reserved, occupied, or clean.

+showTableLayout() : bool

//Displays the arrangement of tables in the restaurant

+showTableStatus(in table:Table) : string

//Returns the status of a table

ManagerInterface

Attributes:

Operations:

+editEmployee(in employee:Employee, in time:string) : void

//Add ore remove employees

+editTable(in table:Table, in action:string) : void

//Change all aspects of tables including location and status

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

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

+viewEmployeeInformation(in employee:Employee) : bool

//Shows the profile of an employee

+viewPayrollInformation(in employee:Employee) : bool

//Shows the payroll of an employee

+showTableLayout(in table:Table) : bool

//Displays the arrangement of tables in the restaurant

UserInterface

Attributes:

Operations:

+renderInterface(in type:string)

//Creates the screen for the user interfaces

c. Traceability Matrix

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

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

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

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

	Software Classes												
Domain Concepts	Employee	Clock	EmployeeControl	PayrollControl	Order	FoodItem	FoodProgress	OrderControl	StockItem	StockStats	InventoryControl	Reservation	Table
Employee	x												
Clock		x											
ClockOutRequest		x											
ClockInRequest		x											
AddEmployeeRequest			x										
RemoveEmployeeRequest			x										
UpdateEmployeeRequest			x										
ReleasePayment				x									
Order					x								
MenuItem						x							
UpdateOrderStatusRequest							x						
EditOrderRequest								x					
AddOrderRequest								x					
CancelOrderRequest								x					
InventoryItem									x				
ViewInventoryRequest										x			
AddInventoryItemRequest										x			
RemoveInventoryItemRequest										x			
AddInventoryRequest											x		
RemoveInventoryRequest											x		
Reservation												x	
Table													x
EditLayoutRequest													
ViewLayoutRequest													
ViewTableStatusRequest													
EditTableStatusRequest													
ViewStatisticInformation													
ReservationRequest													
PayBillRequest													
ViewBillRequest													
ViewPayrollRequest													
UserInterface													
Menu													
AddMenuItemRequest													
RemoveMenuItemRequest													
Controller													
DatabseConnection													
Archiver													

Table 2.1a: Part of the traceability matrix relating classes to the domain concepts.

	Software Classes												
Domain Concepts	TableControl	StatsControl	BusBoyInterface	CustomerInterface	ManagerInterface	HostInterface	StatInterface	UserInterface	Menu	MenuControl	MainController	DBConnection	Archiver
Employee													
Clock													
ClockOutRequest													
ClockInRequest													
AddEmployeeRequest					x								
RemoveEmployeeRequest					x								
UpdateEmployeeRequest					x								
ReleasePayment					x								
Order													
MenuItem													
UpdateOrderStatusRequest													
EditOrderRequest					x								
AddOrderRequest					x								
CancelOrderRequest					x								
InventoryItem													
ViewInventoryRequest							x						
AddInventoryItemRequest													
RemoveInventoryItemRequest													
AddInventoryRequest													
RemoveInventoryRequest													
Reservation													
Table						x							
EditLayoutRequest	x				x	x							
ViewLayoutRequest	x				x	x							
ViewTableStatusRequest	x		x		x	x							
EditTableStatusRequest	x		x		x	x							
ViewStatisticInformation		x			x								
ReservationRequest				x									
PayBillRequest				x									
ViewBillRequest				x									
ViewPayrollRequest				x									
UserInterface								x					
Menu									x				
AddMenuItemRequest										x			
RemoveMenuItemRequest										x			
Controller											x		
DatabseConnection												x	
Archiver													x

Table 2.1b: Part of the traceability matrix relating the classes and domain concepts.

IV. System Architecture and System Design

a. Architectural Styles

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

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

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

b. Identifying Subsystems

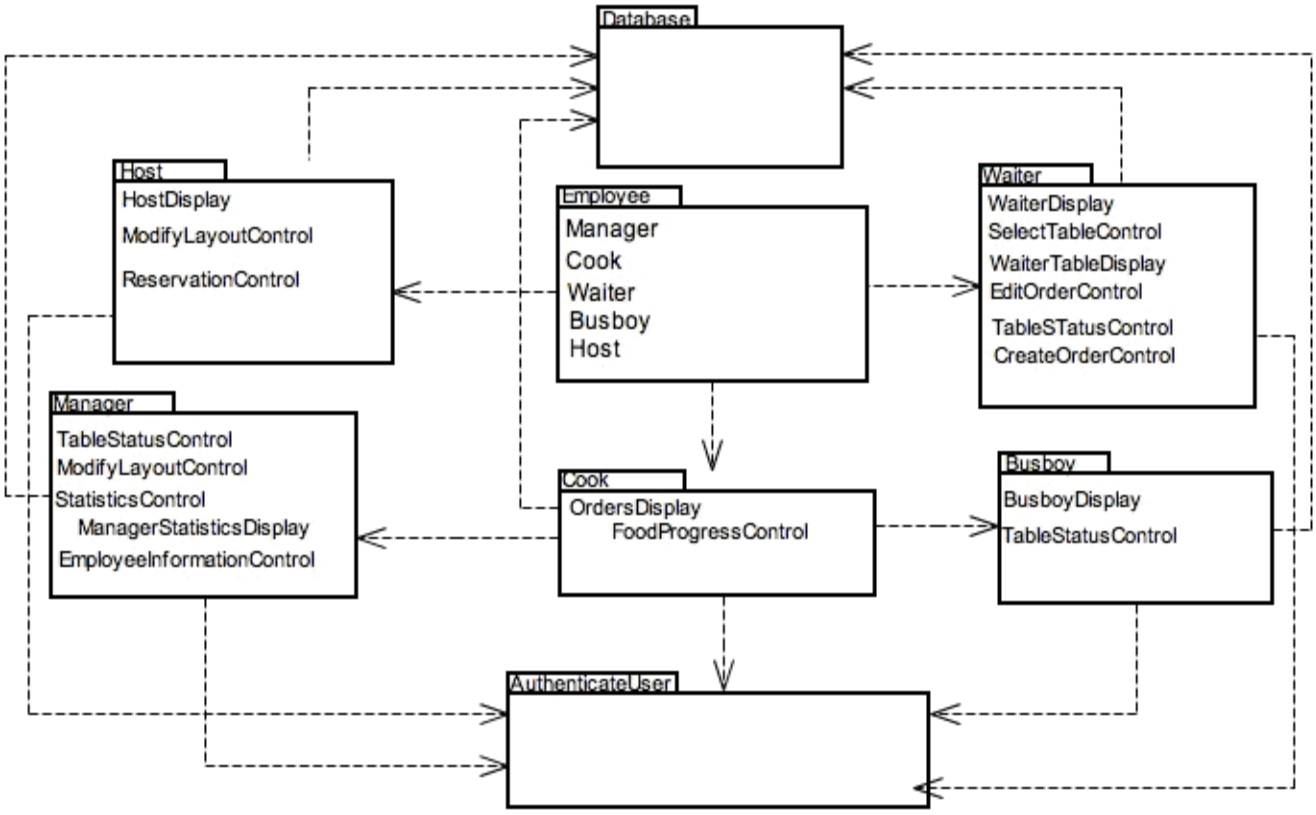

Figure 2.8: Figure showing the subsystems of the entire system.

b. Mapping Subsystems to Hardware

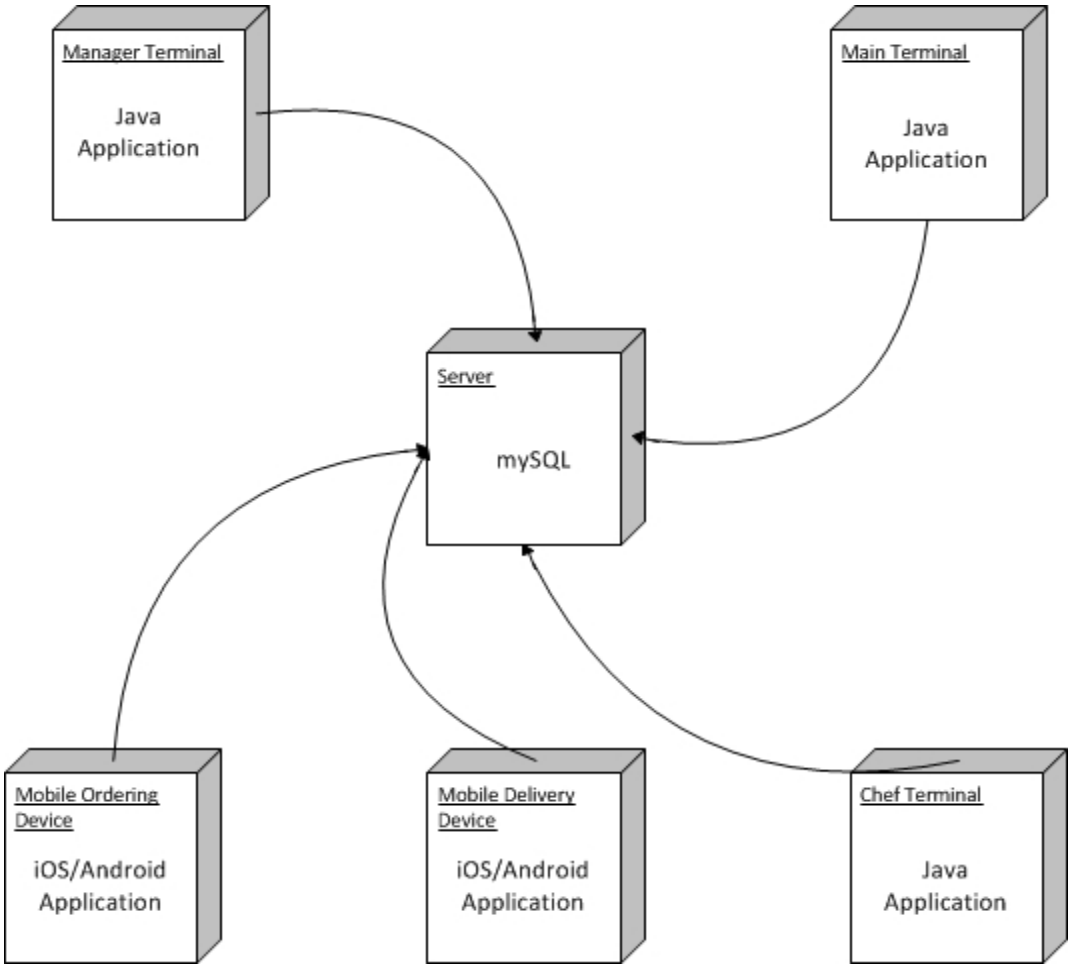

Figure 2.9: Diagram showing the connection between the subsystem and hardware.

c. Persistent Data Storage

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

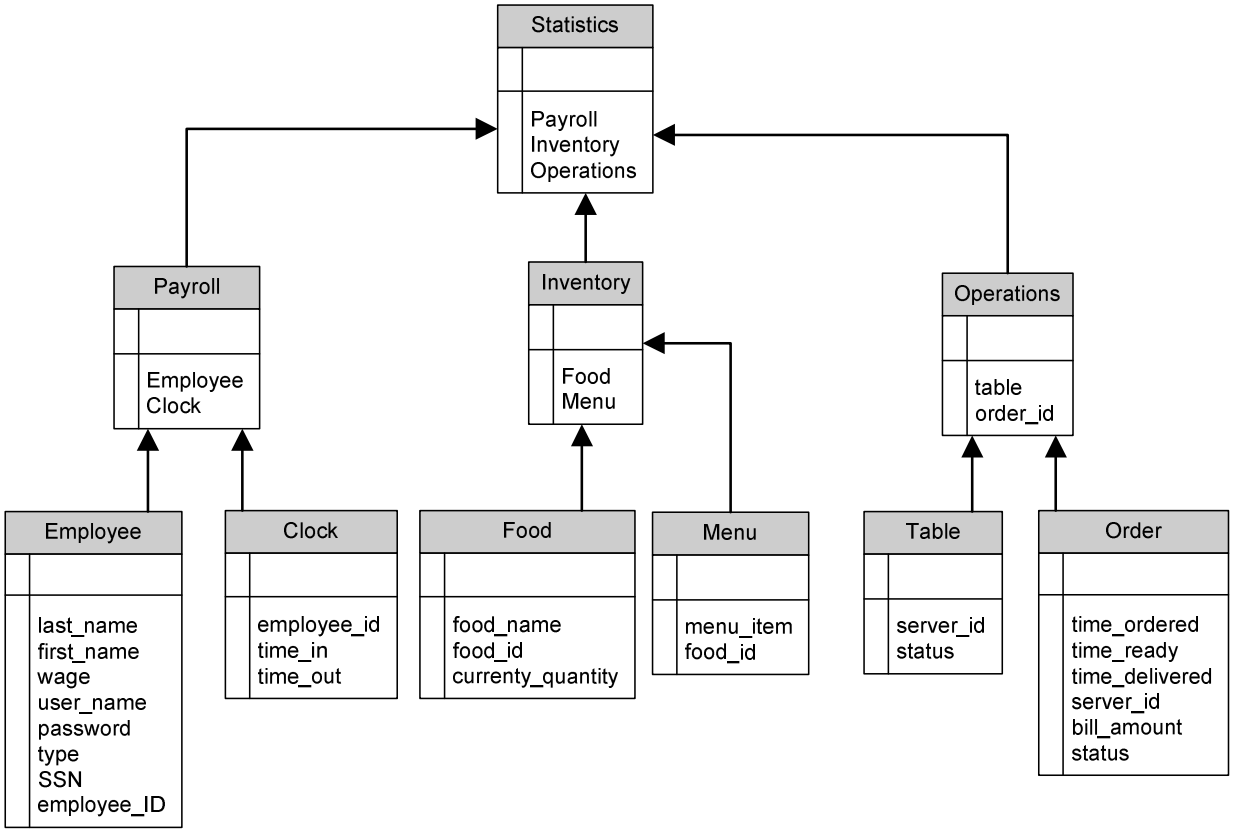

Figure 2.10: Diagram showing how data is stored in the system.

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

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

e. Network Protocol

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

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

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

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

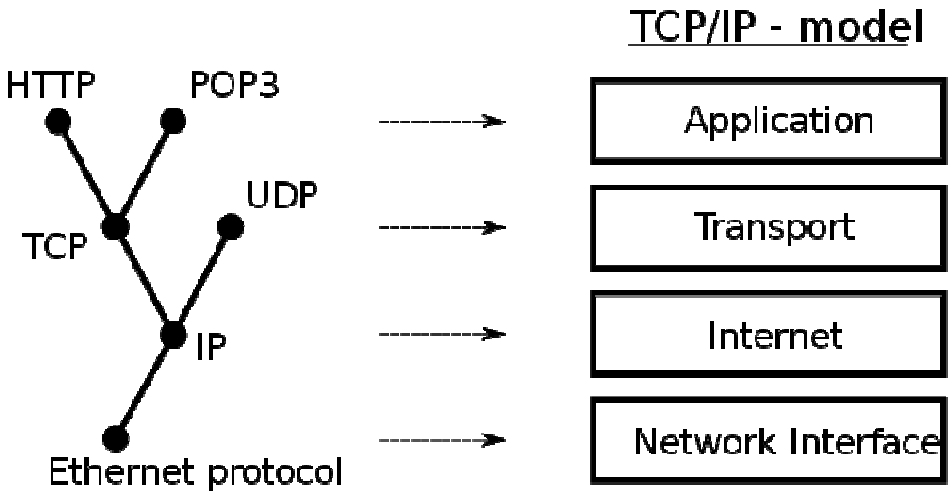

Figure 2.11: The network protocal show with connection to the TCP/IP.

f. Global Control Flow

Execution Orderness

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

Time Dependency

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

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

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

Concurrency

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

g. Minimum Hardware Requirements

Server

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

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

Table 2.2: The minimum hardware requirements for the server.

Hand Held Devices

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

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

Table 2.3: The minimum hardware requirements for hand held devices

Desktop Client

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

Hardware	Minimum Requirement
Processor	1 GHz
RAM	512 MB
Hard Drive Space	5 GB
Network	10/100/1000 NIC Wifi 802.11n
Screen	Size: 20" Resolution: VGA(640x480)

Table 2.4: The minimum hardware requirements for the desktop client.

V. Algorithms and Data Structures

a. Algorithms

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

Check Stats/Stock

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

  Menu trends require the user (manager) to input a given time range and (optionally) a given item(s). The request is checked to be valid, and if it is, the process continues. Whenever an item is ordered, the stats database value under the current date and under the subcategory of the specific item sold is incremented, so the system simply queries the number under the category of the dates given and under the category of the specified items. Once this data is retrieved, the information is plotted on a line graph, the x axis being time (i.e. days of the week) and the y axis being the number of units sold of said item(s) (i.e. hamburger, salad, etc.).
  Customer trends are a very similar process. When a party of customers is seated the stats database value under the current date and under the subcategory of the current hour is increased by the number of customers in the seated party. The system retrieves this information for a given time range and plots the data on a line graph, the x axis corresponding to time and the y axis corresponding to number of customers. The customer trends is given greater precision than menu trends (hours vs. days, respectively) because it is more important to know specifically what times of the day are busier than others. It is important to know how many items are sold in a given day, but the extra precision (hours instead of days) is unnecessary and only complicates the process.
- Check Stock simply retrieves the current amount of each item in the stock database and displays it as a bar graph. The x axis shows ingredients and the y axis show amount.

Edit Layout

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

b. Data Structures

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

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

The array data structure is useful for holding the status of all the tables in the restaurant. A disadvantage often cited for arrays is that the size of an array is static, but with the restaurant having a constant number of tables with each table being capable of holding a maximum amount of guests, this disadvantage is not an issue. The array structure will hold all the tables. Each table will be a data element with information such as the table status (clean, dirty, or occupied), the maximum amount of people that could

be sat at that table, the bill that is to be assigned to that table, and the server to be assigned to that table. There will be various table sizes in the restaurant. Specifically, our restaurant will be divided into sections with each section being assigned to a particular server. There will be three section formats: one for the AM shift, one for PM shift for Monday to Thursday, and one for the PM shift for Friday to Sunday. The restaurant will consist of 15 tables that could accommodate 4, 11 booths that could accommodate 6, 6 booths that could accommodate 4, 1 booth that could accommodate 8, 1 table that could accommodate 6, and 8 tables that could accommodate 2. The specifics have been taken directly from the Applebee's setup. Details as to how this information was acquired are mentioned in the Interview section. Additionally, when a large party consisting of more than could be accommodated at a single table or booth would require tables toe be moved accordingly. With an array structure, we could loop through the entire array and see if there are contiguous tables where the sum of the maximum amount of people that could be sat is equal or one to two less than the party size. If we were to place two tables together to represent one party, we can simply have these two tables be marked as occupied in the array as well as have both tables be allocated to the same bill.

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

VI. User Interface Design and Implementation

As of right now, our user interface is very intuitive, and thus we have not made any drastic changes to the design.

The implementation of our initial user interface design so far has been fairly straightforward. The first thing that needed to be done was to make a separate file for each individual element of user interface (buttons, logos, etc). Most of these elements had parts that are either clear or opaque. This is where the first problem arises. Most image file formats (.jpeg, .png , .tif, etc) do not preserve clear or opaque backgrounds. After doing some research, however, we found that we could preserve the clear look of our elements by saving them as a .gif image file.

Next, we had to save multiple sizes of each design element. Since we plan on having a mobile platform and a desktop platform we needed to have multiple sizes to fit a wide array of Android devices as well as a size for the desktop platform. Although we do have different image sizes for the design elements for the Android platform, we are implementing a fluid layout. This means that built in to the Android framework is a tool that does most of the thinking for us when it comes to which device uses what size background, what resolution, etc.

VII. Design of Tests

a. Test Cases

Table 2.5: MakeReservation

Test-case Identifier:	TC-1
Use Case Tested:	UC-1, main success scenario
Pass/fail Criteria:	The test passes if the user is successfully able to make a reservation for a date and time where there if less than sixty percent of the tables in the restaurant are reserved.
Input Data:	Date, Time, Party Size
Test Procedure:	**Expected Result:**
Step 1. Select a date, time, and party size for which there are sixty percent or more tables reserved.	System displays the date, time, and party size chosen and then displays a message stating that no more reservations can be made for the specified date, time, and party size.
Step 2. Select a date and time for which there are less than sixty percent or more tables reserved.	System displays the date, time and party size chosen and then prompts the user for information. (Credit Card, Name, Number, Address).

Table 2.6: AddOrder

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

Table 2.7: EditOrder

Test-case Identifier:	TC-3
Use Case Tested:	UC-3, main success scenario
Pass/fail Criteria:	The test passes if the user is successfully able to edit an order before the order has started being prepared.
Input Data:	Food Item
Test Procedure:	**Expected Result:**
Step 1. Select item that has been ordered and has started to be prepared.	System displays information about the item that was ordered.
Step 2. Select edit order.	System displays a message stating that the order has started to be prepared and can no longer be changed or canceled and returns user to menu.
Step 3. Select item that has been ordered and has not yet started to be prepared.	System displays information about the item that was ordered.
Step 4. Select edit order.	System displays a menu to change the items that were ordered.
Step 5. Select options and submit	System displays the changes to be made, prompts the user that the changes have been successfully made and returns the user to the menu.

Table 2.8: CancelOrder

Test-case Identifier: TC-4	
Use Case Tested: UC-4, main success scenario	
Pass/fail Criteria: The test passes if the user is successfully able to cancel an order before the order has started being prepared.	
Input Data: Food Item	
Test Procedure:	**Expected Result:**
Step 1. Select item that has been ordered and has started to be prepared.	System displays information about the item that was ordered.
Step 2. Select cancel order.	System displays a message stating that the order has started to be prepared and can no longer be changed or canceled and returns user to menu.
Step 3. Select item that has been ordered and has not yet started to be prepared.	System displays information about the item that was ordered.
Step 4. Select cancel order and verify cancellation.	System displays a prompt stating the order was successfully cancelled and returns user to menu.

Table 2.9: PayBill

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

Table 2.10: AddInventoryItem

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

Table 2.11: AddInventory

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

Table 2.12: RemoveInventory

Test-case Identifier:	TC-8
Use Case Tested:	UC-14, main success scenario
Pass/fail Criteria:	The test passes if the user is successfully able to remove stock from an inventory item that exists in the database.
Input Data:	Inventory Item, Stock
Test Procedure:	**Expected Result:**
Step 12. Select existing inventory item.	System prompts user to add stock, remove stock, remove item, or cancel.
Step 13. Select remove stock.	System prompts user to enter a value.
Step 14. Enter value greater than current stock and select submit.	System prompts user that entered value is too large and prompts user to enter another value
Step 15. Enter value less than current stock and select submit.	System updates the stock value in the database for the item chosen and returns user to screen with inventory items.

Table 2.13: RemoveInventoryItem

Test-case Identifier:	TC-9
Use Case Tested:	UC-12, main success scenario
Pass/fail Criteria:	The test passes if the user is successfully able to remove an existing inventory item.
Input Data:	Inventory Item
Test Procedure:	**Expected Result:**
Step 16. Select existing inventory item.	System prompts user to add stock, remove stock, remove item, or cancel.
Step 17. Select remove item	System prompts user to confirm removal of item.
Step 18. Select confirm.	System removes the item from the inventory database and displays a message to the user stating that the item was successfully removed and returns the user to the list of inventory items.

Table 2.14: ClockIn

Test-case Identifier:	TC -10
Use Case Tested:	UC-23, main success scenario
Pass/fail Criteria:	The test passes if a timer starts for the user who clocks in upon the start of work.
Input Data:	Click

Test Procedure:	**Expected Result:**
Step 1. Click clock out while not clocked in.	System prompts that user must be clocked in to clock out.
Step 2. Click clock in.	System starts a timer.

Table 2.15: ClockOut

Test-case Identifier:	TC -11
Use Case Tested:	UC-24, main success scenario
Pass/fail Criteria:	The test passes if the user stops the timer and clocks out upon finishing a work shift.
Input Data:	Click

Test Procedure:	**Expected Result:**
Step 1. Click clock in while still clocked in. Step 2. Click clock out.	System prompts that the user is already clocked in. System stops the timer; records total hours worked; adds hours worked to the user's monthly hours worked.

Table 2.16: UpdateEmployee

Test-case Identifier:	TC -12
Use Case Tested:	UC-22, main success scenario
Pass/fail Criteria:	The test passes if the user inputs valid employee information.
Input Data:	FirstName, LastName, Wage, Username, Password, Type, SSN
Test Procedure:	**Expected Result:**
Step 1. Replace old FirstName and type in an invalid new FirstName	System places a "✗" next to the text field.
Step 2. Replace old FirstName with a valid new FirstName	System places a "✓" next to the text field.
Step 3. Replace old LastName with an invalid new LastName	System places a "✗" next to the text field.
Step 4. Replace old LastName with an valid new LastName	System places a "✓" next to the text field.
Step 5. Wage is changed to an invalid input.	System places a "✗" next to the text field.
Step 6. Wage is changed to a positive number.	System places a "✓" next to the text field.
Step 7. Replace Username with an invalid new Username.	System places a "✗" next to the text field.
Step 8. Replace Username with a valid new Username	System places a "✓" next to the text field.
Step 9. Replace Password with an invalid new Password	System places a "✗" next to the text field.
Step 10. Replace Password with a valid new password.	System places a "✓" next to the text field.
Step 11. Change Type to an invalid occupation.	System places a "✗" next to the text field.
Step 12. Change Type to a valid occupation.	System places a "✓" next to the text field.
Step 13. Change SSN to an invalid number.	System places a "✗" next to the text field.
Step 14. Change SSN to a valid number	System places a "✓" next to the text field.

Table 2.17: AddEmployee

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

Table 2.18: RemoveEmployee

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

Table 2.19: AddMenuItem

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

Table 2.20: RemoveMenuItem

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

Table 2.21: EditLayout

Test-case Identifier:	TC -17
Use Case Tested:	UC-6, main success scenario
Pass/fail Criteria:	The test passes if the user moves a table to a new coordinate with no obstructions and combines with another table when put within its vicinity
Input Data:	Click and drag
Test Procedure:	**Expected Result:**
Step 1. Click and drag a table on top of another table.	System returns dragged table to its last saved position; prompts user of an obstruction.
Step 2. Click and drag a table to a place not within the radius of other tables and objects.	System records new coordinates; table moved and saved to new location.
Step 3. Click and drag a table to the edge of another table.	System records new coordinates; table merged with the adjacent table to be treated as one table.

b. Unit Tests

i. Table

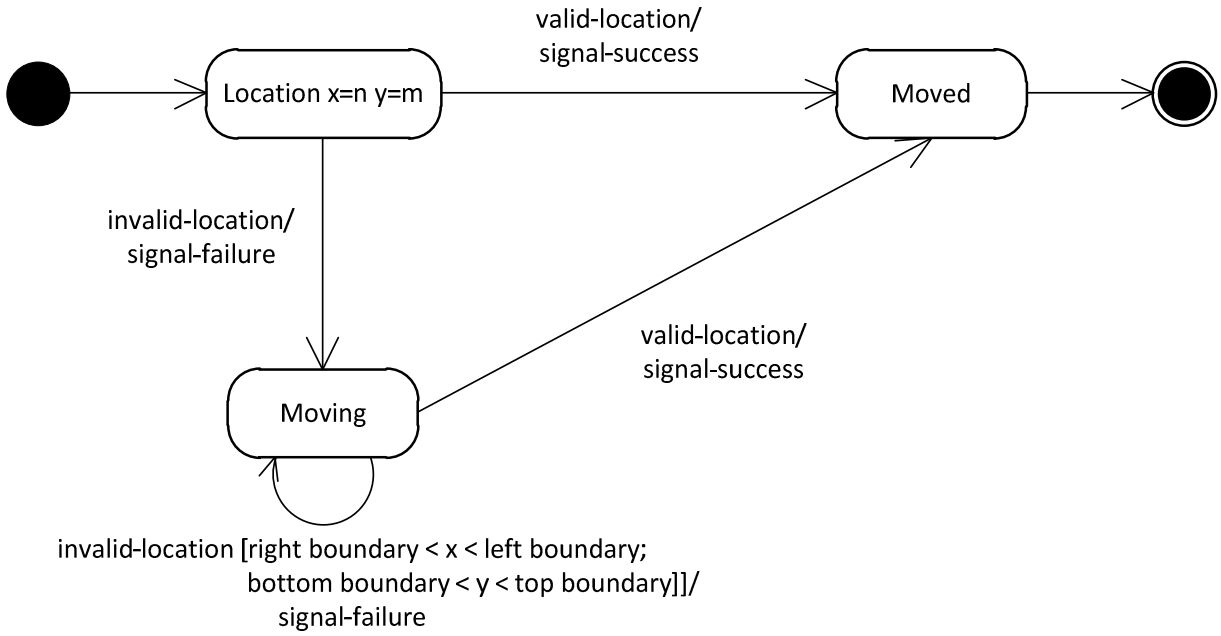

Figure 2.13: State diagram of the Table Class.

Method calls to test all states and transitions
Table A, B; // First tables are made.

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

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

ii. FoodProgress

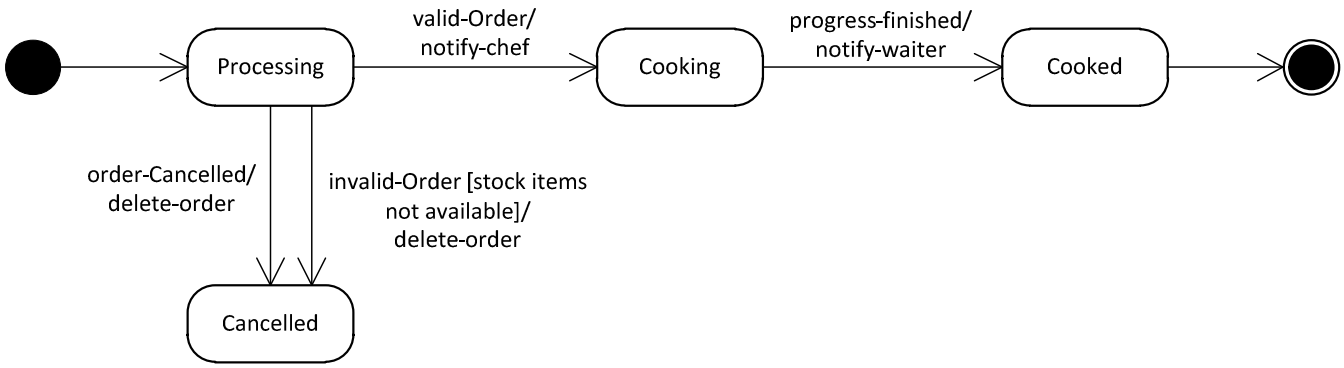

Figure 2.14: State diagram of the FoodProgress Class.

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

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

First, all cancellations from a processing order are tested. This is done with the first methods calls with invalidOrder. If the method returns true, then the order is valid. However, if the ordered item is not a foodItem on the menu or if there is not enough stocks in the inventory to make the ordered food, the order will be cancelled by the system and the waiter notified about the situation. Also, if the customer decided to cancel an order, it will be allowed, since cooking the order has not been started.
Once a foodItem is processed, the Order is added to the queue of foods for the chef to cook. Once the chef gets to the specified food item, he/she changes the status of progress to cooking. This is seen in the transition from processing to the state of cooking. When the chef is done cooking, the order status is set to finished, representing the final state of cooked.

iii. DBConnection

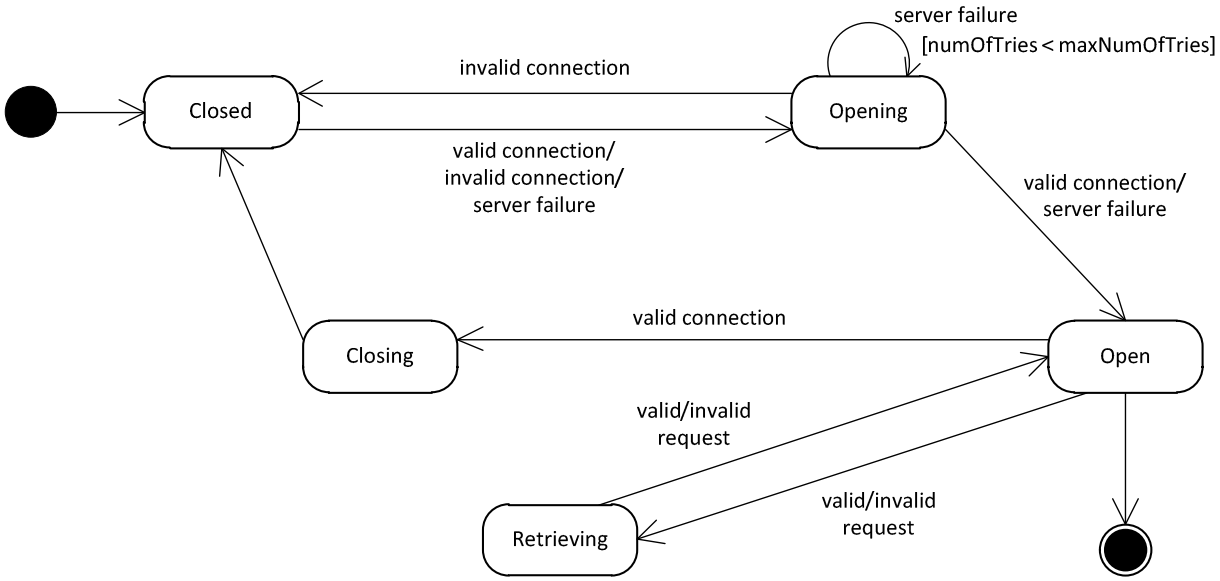

Figure 2.15: State diagram of the DBConnection class.

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

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

iv. Employee

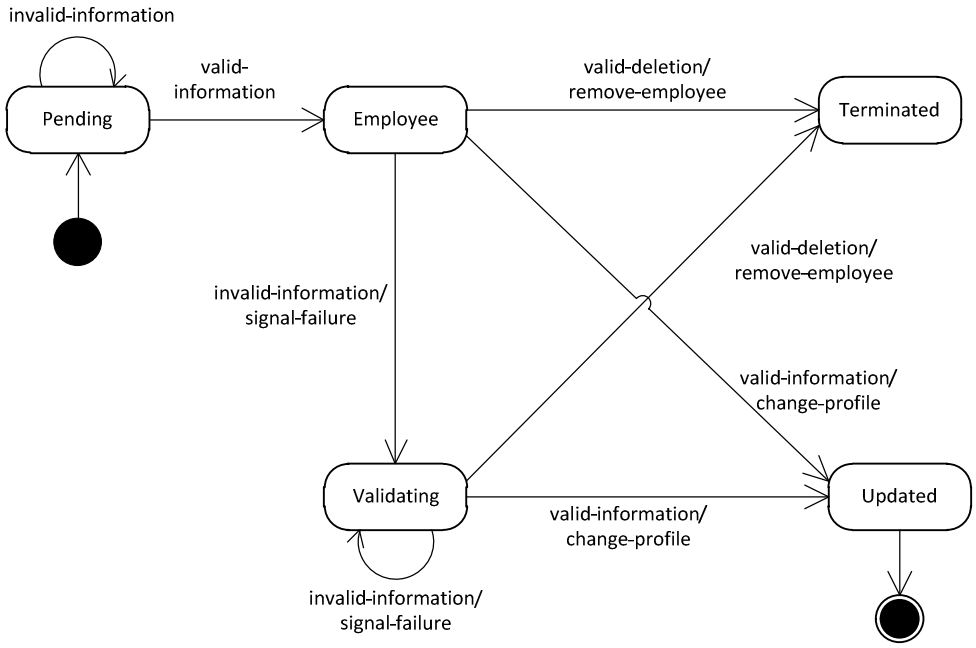

Figure 2.16: State diagram of the Employee class.

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

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

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

transitions from the state Employee to Terminated. The state can go from Employee to Validating if there is a signal failure and if a valid connection is established the state proceeds to Terminated.

c. Integration Testing

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

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

VIII. Interview Questions

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

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

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

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

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

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

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

2. Why would an employee get fired?
An employee would get fired if they weren't following the policies of the company, which are outlined in the employee handbook. Also, if an employee does not show up to

work and does not have proper coverage or medical note, that employee is automatically fired. Also, if an employee is giving out free food or beverages without manager knowledge.

IX. Project Management and Plan of Work

a. Merging the Contributions From Individual Team Members

For this report, Jazmin Garcia, Vishal Shah, and Damon Chow collectively took turns merging the contributions of each group member. Each assured that the report was uniform in formatting and appearance.

The only issues met were the issues of fixing report 1 and appropriately coordinating those changed to this report. Vishal Shah and Damon Chow who were initially responsible for both in the first report fixed the domain model as well as the class diagram. Once these changes were completed, we were able to assign responsibilities according to the changes.

b. Progress Coordination and Progress Report

To manage such a large process, we found it necessary for the team to be split into two groups: one team for mobile app development and one team for main app/database development. Some team members had greater strengths in certain areas, so splitting up the tasks allowed for a more efficient development process. However, all team members were involved on major decisions that concerned the entire system, and simply specified in a certain area.

We started development by focusing on the most important aspects of the application. The main application was started first, and so far allows the user to clock in/out and place an order. This covers the use cases ClockIn, ClockOut, PlaceOrder, EditOrder, and CancelOrder. Also, the user can edit the menu items, covering the use cases AddToMenu and DeleteFromMenu. Development on the mobile application has also begun. So far, the user can place an order, and we are currently working on the communication between the mobile app and the main app.

We are currently working the restaurant statistics aspect of the application that allows the user to view stock, menu trends, customer trends, and payroll. We felt this was the most logical progression to take as it goes in order of priority. That is, the application is functional for basic operation and more advanced features are added on once the basic ones are accomplished.

c. Plan of Work

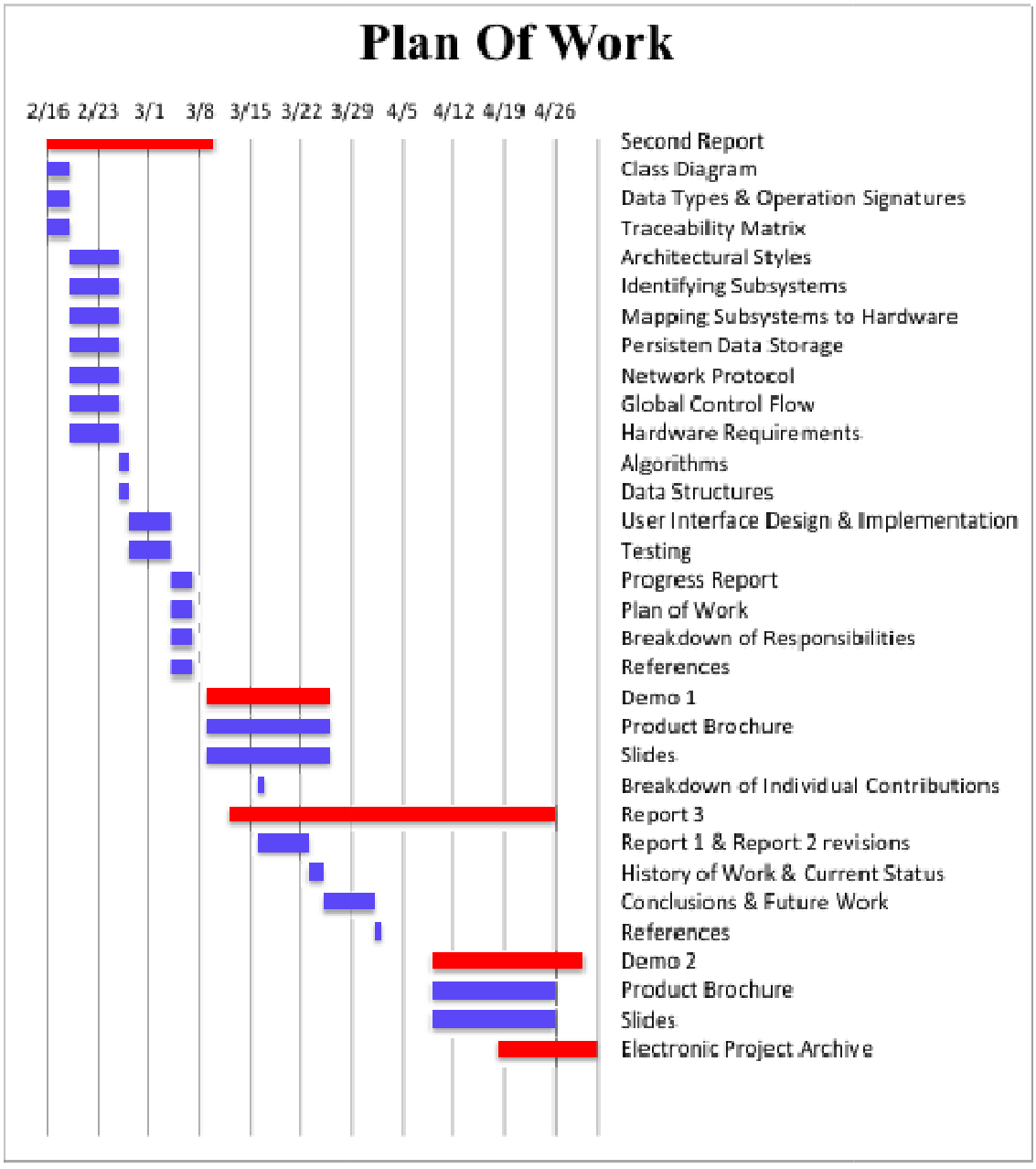

Figure 2.17: The Gantt Chart of the plan of work.

Second Report	Start Date	Days till Due Date	End Date
Class Diagram	2/17/12	23	3/9/12
Data Types & Operation Signatures	2/17/12	3	2/20/12
Traceability Matrix	2/17/12	3	2/20/12
Architectural Styles	2/17/12	3	2/20/12
Identifying Subsystems	2/20/12	7	2/27/12
Mapping Subsystems to Hardware	2/20/12	7	2/27/12
Persistent Data Storage	2/20/12	7	2/27/12
Network Protocol	2/20/12	7	2/27/12
Global Control Flow	2/20/12	7	2/27/12
Hardware Requirements	2/20/12	7	2/27/12
Algorithms	2/20/12	7	2/27/12
Data Structures	2/27/12	1	2/28/12
User Interface Design & Implementation	2/27/12	1	2/28/12
Testing	2/29/12	6	4/5/12
Progress Report	2/28/12	6	4/5/12
Plan of Work	3/5/12	3	3/8/12
Breakdown of Responsibilities	3/5/12	3	3/8/12
References	3/5/12	3	3/8/12
Demo 1	3/5/12	3	3/8/12
Product Brochure	3/10/12	17	3/27/12
Slides	3/10/12	17	3/27/12
Breakdown of Individual Contributions	3/10/12	17	3/27/12
Report 3	3/17/12	1	3/18/12
Report 1 & Report 2 revisions	3/13/12	45	4/27/12
History of Work & Current Status	3/17/12	7	3/24/12
Conclusions & Future Work	3/24/12	2	3/26/12
References	3/26/12	7	4/2/12
Demo 2	4/2/12	1	4/3/12
Product Brochure	4/10/12	21	5/1/12
Slides	4/10/12	17	4/27/12
Electronic Project Archive	4/10/12	17	4/27/12
	4/13/12	14	5/3/12

Figure 2.18: The date related to the Gantt Chart.

Our group retained the same Plan of Work as in the First Report. The Gannt Chart is above with the Tasks listed as well as the date when which a particular task needs to be begun and when the task needs to be completed by. The Red bars indicated one item while the bars in purple indicate a sub item. For example, Report 2 would be considered an item while a sub item would be Algorithms.

d. Breakdown of Responsibilities

Project Manager
Jazmin Garcia
• 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:
• Every one will contribute equally to the coding part of the project.

Lead Developer
Vishal Shah & *Eric Gilbert*
• 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:
• Every one will contribute equally to the coding part of the project.

System Analyst
Greg Paton
• Interaction Diagrams
• System Architecture and System Design (Minor Role)
• Database Design (Minor Role)
• Visio Expert
• Meeting Notes Taker
• Logistics (Minor Role)

Development Assignments:
• Every one will contribute equally to the coding part of the project.

Developer
Damon
• 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:
• Every one will contribute equally to the coding part of the project.

In terms of classes, we will have the following classes assigned to the following team members. Each team member is currently responsible for developing, coding, and testing. These are the main classes we are focusing on although there will be others we will be implementing later on.

1. Employee – *Greg Paton*
2. Order – *Vishal Shah*
3. Menu – *Jazmin Garcia*
4. Reservation – *Eric Gilbert*
5. HostInterface – *Jazmin Garcia*
6. ManagerInterface – *Eric Gilbert*
7. CustomerInterface – *Damon Chow*
8. BusBoyInterface – *Greg Paton*
9. Table – *Jazmin Garcia*
10. InventoryControl – *Vishal Shah*
11. PayrollControl – *Vishal Shah*
12. FoodProgress – *Damon Chow*

If not clear before, the group will perform integration testing collectively once individual components are completed.

X. References

"Concurrency Control." *Wikipedia*. Wikimedia Foundation, 03 Nov. 2012. Web. 12 Mar. 2012.

<http://en.wikipedia.org/wiki/Concurrency_control>.

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

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

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

"Real-time Simulation." *Wikipedia*. Wikimedia Foundation, 21 Jan. 2012. Web. 12 Mar. 2012.

<http://en.wikipedia.org/wiki/Real-time_Simulation>.