Restaurant/Menu GUI

Project Report

By: Benjamin Ramos, Zachary Montoya,

Casey Mackin, Ryan Carnaghi

Problem Statement:

In the real world when you go to a nice restaurant the customer spends most of the time sitting and ordering food from a waiter that will not always be around. Or sometimes you take a very long time thinking about what you want and the customer then feels pressured to order something even though it may not be what they wanted. This can make some customers angry or displeased with the service, so our group wanted to come up with a solution to take out the time waiting for a waiter to serve you or one that pressures you to order food. Our solution to this problem was to create a GUI menu in which the customer can add his order to something like a shopping cart on an internet shopping site where he can then send his order directly to the kitchen instead of waiting for the waiter to come by. This will make the customer more comfortable and overall happier, and, in turn, the waiters will have less work by only having to seat the customers and bringing the food to the table.

In our proposal we initially said that there were 5 key classes: Customer, Waiter, Table, Food, Categories. As we worked on the project we found that the Waiter class would just be directly part of our GUI and would not need a class. Also the Categories class in our project proposal has been taken out and now instead we created text files already categorized by different types of foods and then added them to the GUI menu using a driver file.

System Boundary:

For this project we mainly wanted to create a menu that would be able to create a list of food that would be separated into different categories such as drinks, entrees, appetizers, and desserts. From here the customer would be able to see the description of the food/drink and also the price and then choose accordingly from that. The customer can then also add more food and drinks later to the order. They can also pay for the food whenever they want. We also wanted to create an easy way for waiters to know which tables were occupied by customers and also to seat new customers. This would also help to seat the customers to the right size of table.

For this project we excluded the element of time so everything will be instantaneous. In other words, as soon as someone orders food they will immediately get it sent to their table. We also excluded a cook class, which would cook the food and then give the food to the waiters to be sent to the tables.

Domain Analysis:

The overall system will consist mainly of six classes: Restaurant, Waiter, Customer, Table, Menu, and Food. Restaurant as shown below will be associated with the Waiter and Table class, where many Waiters work for the Restaurant and many Tables are in the Restaurant. The Waiter class will server many Customers and work for only one Restaurant. The Customer will be served by only one Waiter, and the Customer will only sit at one Table, as well as only order from one menu. The Table class will sit none or many Customers and the Table has a menu from which the customer can order from. The Menu class will be a class that is aggregated to each table and The Table class will have many foods and drinks to order. The Food class is also aggregated to the menu.

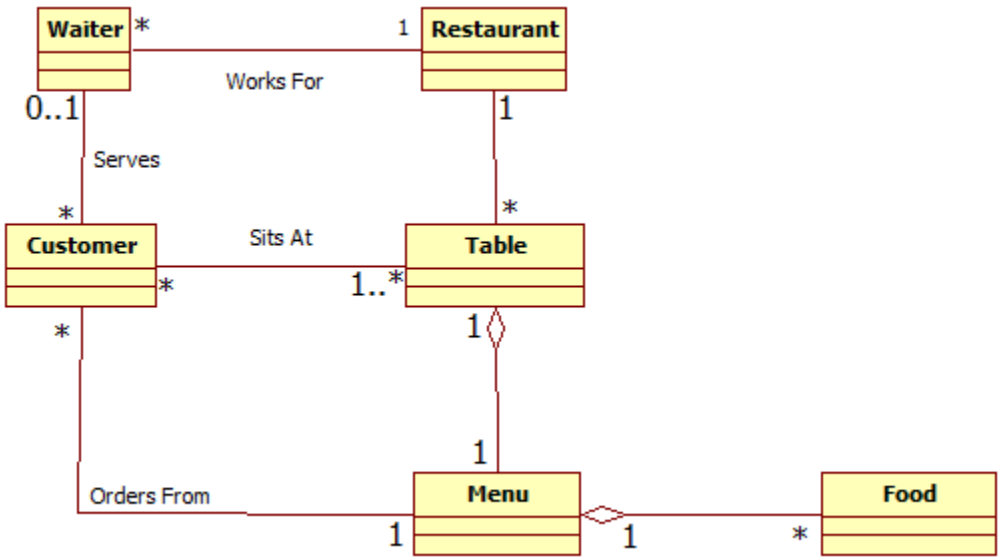


Figure 1- class model diagram with asscociations and aggregations

From the initial group of class we found out that the Waiter class was irrelevant to the GUI so instead of making a Waiter class, we just implemented its method to sit Customers directly into the GUI. The same is also true for the menu class, where the menu class would be a list of different types of food and drinks. This got broken up into several different ArrayList instance variable of the Restaurant Class. With that said, we now only have four key classes: Restaurant, Food, Customer, and Table. From the initial proposal, the Customer class had the most methods. This changed to accommodate the GUI, now only having the getters, setters and a method called isAtTable(), which determines if the customer is at the table. The Table class got many more methods and attributes since the proposal. This includes a Jbutton for the GUI, orderList, and buyList. The methods that were added consist of an addItem method, which will add food/drink items to the orderList, a removeItem method that does just the opposite of the addItem method, a leaveTable method which lets you leave the table as long as you do not have anything in your buyList, a placeOrder method which lets you add one item at a time from the orderList to the buyList, and finally a payOrder method which (for now) just removes items from the buyList since the Customer will have an infinite amount of money. The Food class will basically create the food objects that will be stored in the Restaurant class and displayed in the menus. The Restaurant class will run the GUI. This class will, as stated before, have the lists of different foods and drinks that would be in the menu class we initially thought of.

Although much emphasis is placed on the four classes Restaurant, Food, Table and Customer, a great amount of work was put into the GUI's underlying classes. The MainFrame object extends JFrame and thus acts as the frame that holds everything else. Inside the MainFrame object, a JTabbedPane is used to create a tabbed system within the frame. These tabs will either hold the WaiterPanel or the CustomerPanel. Each panel holds all of the necessary components to correspond with the corresponding actor. However, in our implementation there exists only one Waiter tab, which handles all of the seating, whereas there can exist multiple CustomerPanel tabs, each corresponding to an occupied table. Each of these panels contains numerous components, each with their own listeners where necessary, so describing all of the components is not needed. Below is a UML diagram for the entire system: system, actors, and GUI components.

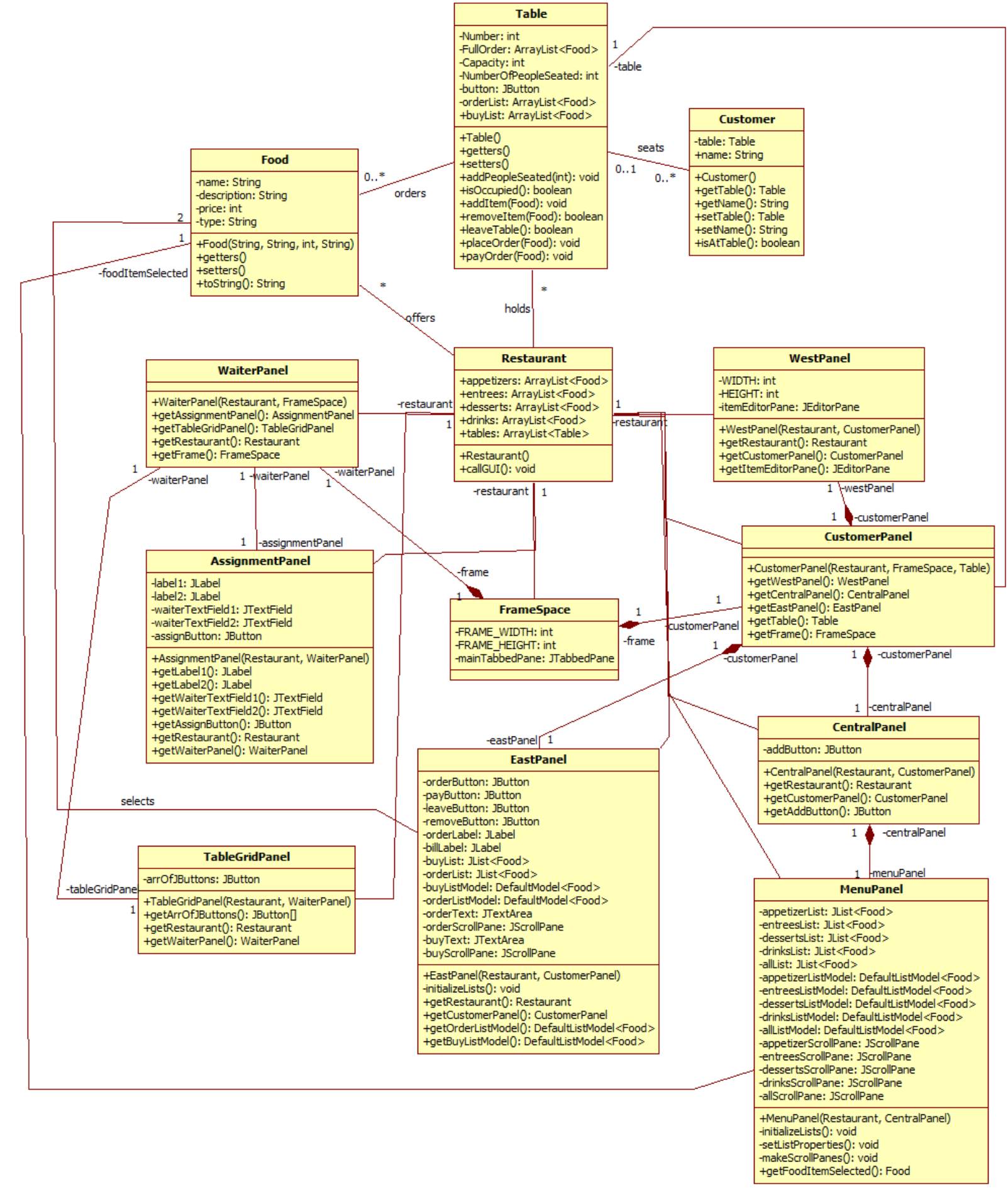
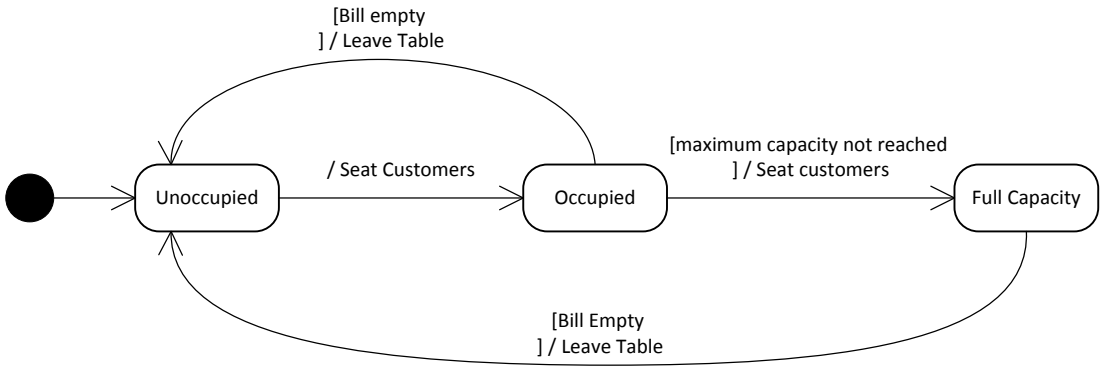


Table
-Number: int
-FullOrder: ArrayList<Food>
-Capacity: int
-NumberOfPeopleSeated: int
-button: JButton
-orderList: ArrayList<Food>
+buyList: ArrayList<Food>
+Table()
+getters()
+setters()
+addPeopleSeated(int): void
+isOccupied(): boolean
+addItem(Food): void
+removeItem(Food): boolean
+leaveTable(): boolean
+placeOrder(Food): void
+payOrder(Food): void
Customer
-table: Table
+name: String
+Customer()
+getTable(): Table
+getName(): String
+setTable(): Table
+setName(): String
+isAtTable(): boolean
Food
-name: String
-description: String
-price: int
-type: String
+Food(String, String, int, String)
+getters()
+setters()
+toString(): String
Restaurant
+appetizers: ArrayList<Food>
+entrees: ArrayList<Food>
+desserts: ArrayList<Food>
+drinks: ArrayList<Food>
+tables: ArrayList<Table>
+Restaurant()
+callGUI(): void
WaiterPanel
+WaiterPanel(Restaurant, FrameSpace)
+getAssignmentPanel(): AssignmentPanel
+getTableGridPanel(): TableGridPanel
+getRestaurant(): Restaurant
+getFrame(): FrameSpace
WestPanel
-WIDTH: int
-HEIGHT: int
-itemEditorPane: JEditorPane
+WestPanel(Restaurant, CustomerPanel)
+getRestaurant(): Restaurant
+getCustomerPanel(): CustomerPanel
+getItemEditorPane(): JEditorPane
CustomerPanel
+CustomerPanel(Restaurant, FrameSpace, Table)
+getWestPanel(): WestPanel
+getCentralPanel(): CentralPanel
+getEastPanel(): EastPanel
+getTable(): Table
+getFrame(): FrameSpace
AssignmentPanel
-label1: JLabel
-label2: JLabel
-waiterTextField1: JTextField
-waiterTextField2: JTextField
-assignButton: JButton
+AssignmentPanel(Restaurant, WaiterPanel)
+getLabel1(): JLabel
+getLabel2(): JLabel
+getWaiterTextField1(): JTextField
+getWaiterTextField2(): JTextField
+getAssignButton(): JButton
+getRestaurant(): Restaurant
+getWaiterPanel(): WaiterPanel
FrameSpace
-FRAME_WIDTH: int
-FRAME_HEIGHT: int
-mainTabbedPane: JTabbedPane
CentralPanel
-addButton: JButton
+CentralPanel(Restaurant, CustomerPanel)
+getRestaurant(): Restaurant
+getCustomerPanel(): CustomerPanel
+getAddButton(): JButton
EastPanel
-orderButton: JButton
-payButton: JButton
-leaveButton: JButton
-removeButton: JButton
-orderLabel: JLabel
-billLabel: JLabel
-buyList: JList<Food>
-orderList: JList<Food>
-buyListModel: DefaultModel<Food>
-orderListModel: DefaultModel<Food>
-orderText: JTextArea
-orderScrollPane: JScrollPane
-buyText: JTextArea
-buyScrollPane: JScrollPane
+EastPanel(Restaurant, CustomerPanel)
-initializeLists(): void
+getRestaurant(): Restaurant
+getCustomerPanel(): CustomerPanel
+getOrderListModel(): DefaultListModel<Food>
+getBuyListModel(): DefaultListModel<Food>
TableGridPanel
-arrOfJButtons: JButton
+TableGridPanel(Restaurant, WaiterPanel)
+getArrOfJButtons(): JButton[]
+getRestaurant(): Restaurant
+getWaiterPanel(): WaiterPanel
MenuPanel
-appetizerList: JList<Food>
-entreesList: JList<Food>
-dessertsList: JList<Food>
-drinksList: JList<Food>
-allList: JList<Food>
-appetizerListModel: DefaultListModel<Food>
-entreesListModel: DefaultListModel<Food>
-dessertsListModel: DefaultListModel<Food>
-drinksListModel: DefaultListModel<Food>
-allListModel: DefaultListModel<Food>
-appetizerScrollPane: JScrollPane
-entreesScrollPane: JScrollPane
-dessertsScrollPane: JScrollPane
-drinksScrollPane: JScrollPane
-allScrollPane: JScrollPane
+MenuPanel(Restaurant, CentralPanel)
-initializeLists(): void
-setListProperties(): void
-makeScrollPanes(): void
+getFoodItemSelected(): Food
orders
seats
offers
holds
-table
-foodItemSelected
-restaurant
-waiterPanel
-assignmentPanel
-frame
-westPanel
-customerPanel
-centralPanel
-menuPanel
-eastPanel
-tableGridPane
selects

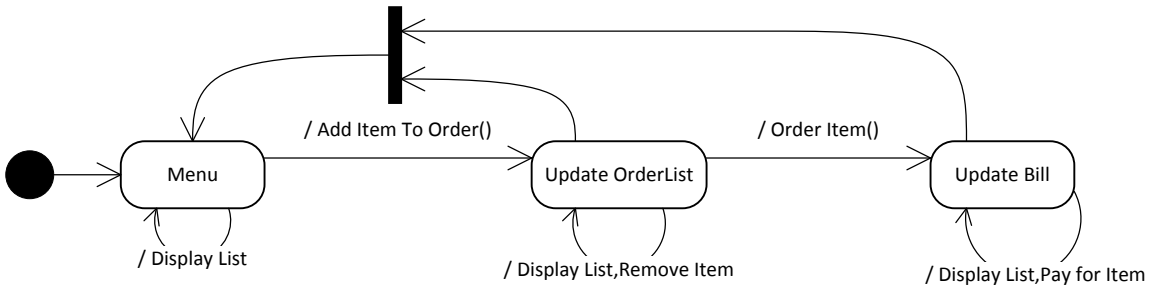
State Models:

In The GUI there is only one class that has states which change when the code is executed: the Table class. It will consist of two separate state models. The first is when you add customers to tables. It will go through three different states. The first state is the unoccupied state where the Table object's occupancy is 0 and the GUI displays green on the grid for that table in the Waiter window. The second state is when customers are added to a table. That table will become Yellow showing that it is occupied. The third state is reached when the table's occupancy reaches the total capacity. No more people can be added and the table becomes red in color.



For the second state model the Table window will have a menu from which the user can order from. When the user adds an item to the order list it will then be displayed on the list of things to be ordered. If something wants to be removed they only have to highlight the item and press the remove button then that item will be removed. Once you are certain you want that item you press the order button to place it in the billing list. Once an item is in the billing list it can only be removed by pressing the pay for item button. Once everything is removed

from the bill the customer may then leave.



Interaction Analysis:

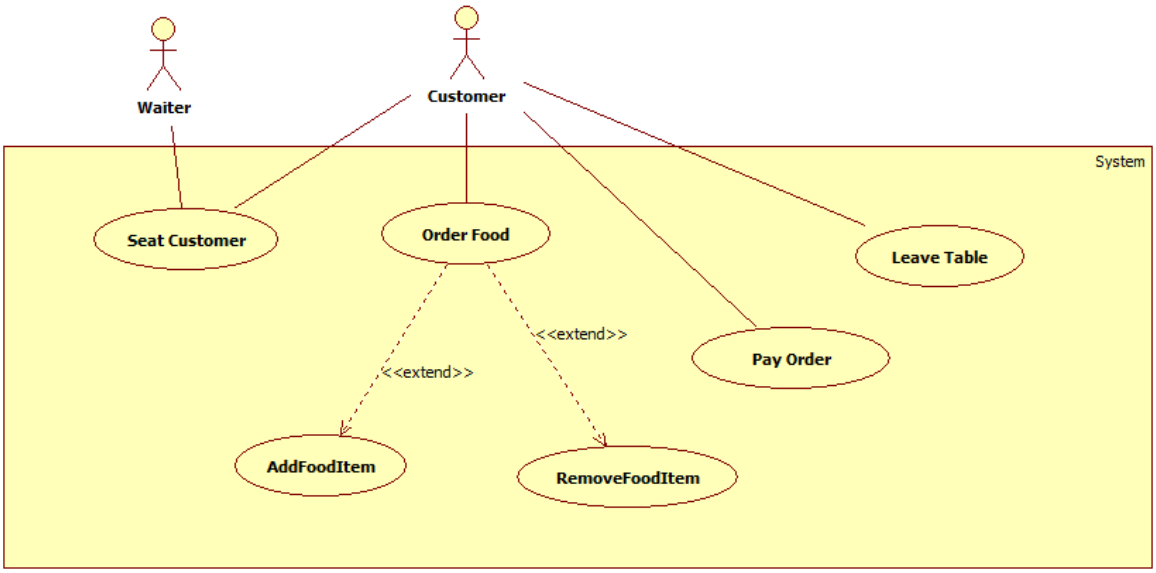


Figure 2 – Use Case Model

A. Uses Cases

Above shows the general uses cases that will be used in the demo. The first of these cases is the seat Customer. Basically the waiter will determine which how many customers there are and then make them sit in the appropriate table. Then the customers will sit at the table that was assigned by the waiter. For the Order Food case the table must be occupied first therefore once seated the customer can order food. This case extends to the cases where one can either add a food item or remove a food item from the list of food to be ordered. Once the food is ordered the item goes to the billing list where the pay order case

then takes effect. Here in the pay order use the customer can highlight food that was ordered and pay for them one-by-one. For the last case leave table the user/ customer will press the leave table button to leave, the table will then check if the bill has been paid for. If it has been paid then the customer may leave otherwise the customer stays at the table.

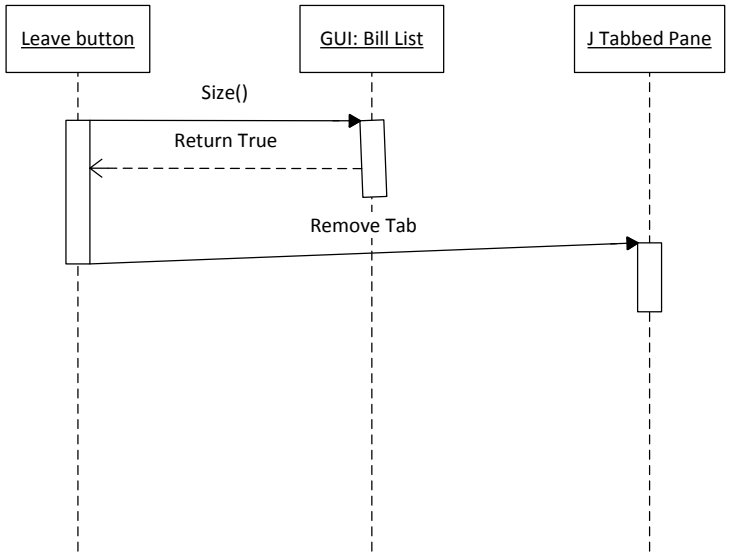


Figure 3 –Sequence diagram of leave table method

B. Boundary cases

For the Boundary cases most of these dealt with the interaction of the customer and the table. One of these cases was seating more customers than the capacity that the table would allow. If this occurred, the user/ waiter will get an error saying that the table is full or that it cannot fit anymore at the table. Another leaving the table as stated in the use cases. If the bill was not paid before leaving then the customer will get an error saying they cannot leave. Another case is paying without anything in the billing. If you try to pay for the check without having anything in the bill section you will not be able to because you need to highlight an item before you can pay for it.

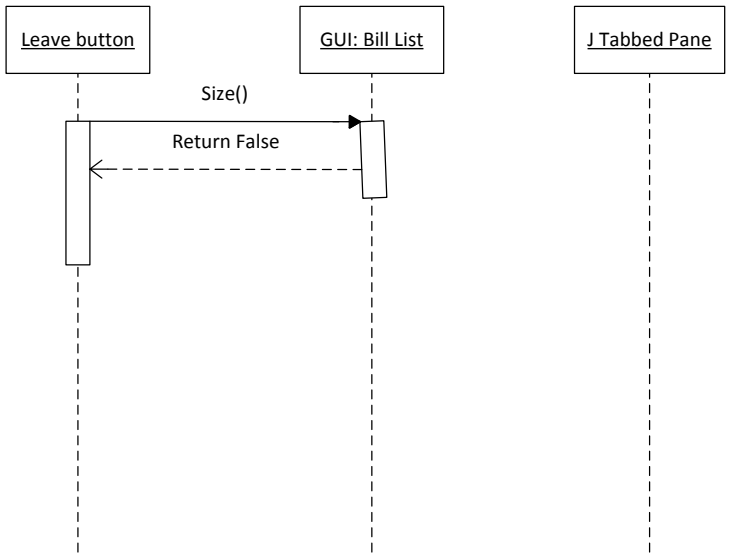


Figure 4 –Sequence Diagram of not leaving the Table

Implementation:

A. Class Implementation

For the implementation of the classes, two of them were directly inserted as part of the GUI functionality. These two classes are the waiter class and the menu class. The waiter class is now a separate window/ tab that is given the ability to seat and check for the occupancy of the table. The menu class is now a menu placed directly into the table tab when one exists. The food class was implemented into the code as a multitude of objects, each containing the information for individual food items present in the text files we scanned through. These files were read through by the driver file. The Customer class in our initial design would have the most methods, but this was not the case with the GUI. Instead, most of its methods and attributes were moved to the Table class. This happened because for the GUI the user would act as the customer when ordering and paying for food. So now the Customer Class is just a place holder to keep track of how many people are sitting at the tables. The Table class has the most interaction in the GUI. When a customer is seated, a tab for the table will show up. This tab will display the menu, ordering list, and the bill. It also has multiple buttons which will let

you add or remove food from your list of items to order along with a pay button and a button to leave the table. The waiter window is the first window you see when you run the program. This is where you can add customers to tables. There is also a grid of the different tables with different capacities. When you click on these different boxes, a window will pop up and show the table number, how many people are sitting there, and what the capacity of that table is.

B. Use Case Implementation

For the cases, we implemented most of them through GUI classes. For seating a customer we added a two JTextField components, one for the amount of people to seat and the other for which table to seat them at. Then you can press the seat button which will take the user input from the JTextField components and place the designated number of Customers at the designated table if doing so would not exceed the table's capacity. The next case is ordering food. We implemented this into the GUI by creating a JList of different food or a menu which will display all the types of food. The food can then be highlighted one at a time and be added to the order list. From the order list, which is also a Jlist, you can highlight the items you added and then either remove them from your list of food to be ordered, or you can order that item which will then be put into another JList called bill. You can perform each of these by clicking the corresponding button underneath the order list. From here the use case pay order can be applied. The item in your bill JList must first be highlighted and then you must press the pay button, which for now will only just remove the item from the list. When you finish paying for food you can press the Jbutton that says "Leave Table". When this button is pressed, it will check the bill JList to make sure it is empty before you can leave, otherwise you will get an error message saying you cannot leave.

Testing & Results:

Initially we tested the code through trial and error runs. Most of them turned out to be successful for what we required the project to do. We tested the JTextField objects to make sure they would not accept characters or negative numbers. If there were any when the button was pressed, a message would pop up telling you that your input was incorrect. If the

input was valid, then it would proceed to add the people to the table and then display, change the table's button from green to yellow or red, depending on the capacity of the table. Below shows the results of these tests:

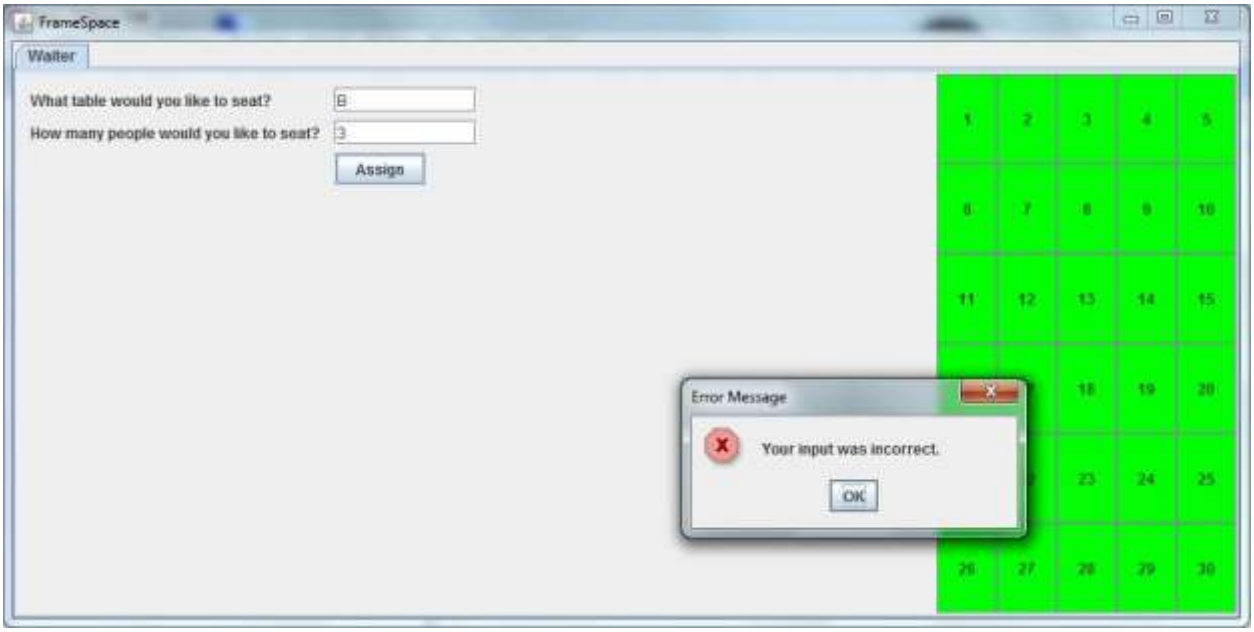


Figure 5- Shows an error message when a character is added.

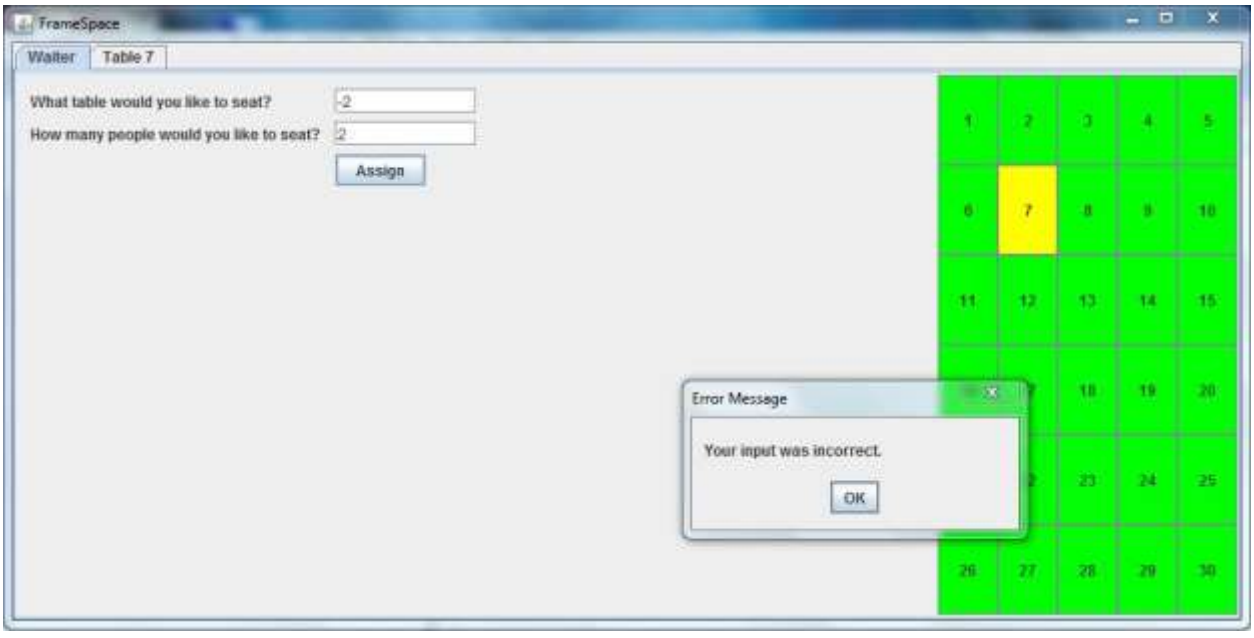


Figure 6 – Figure shows error when a negative number is added

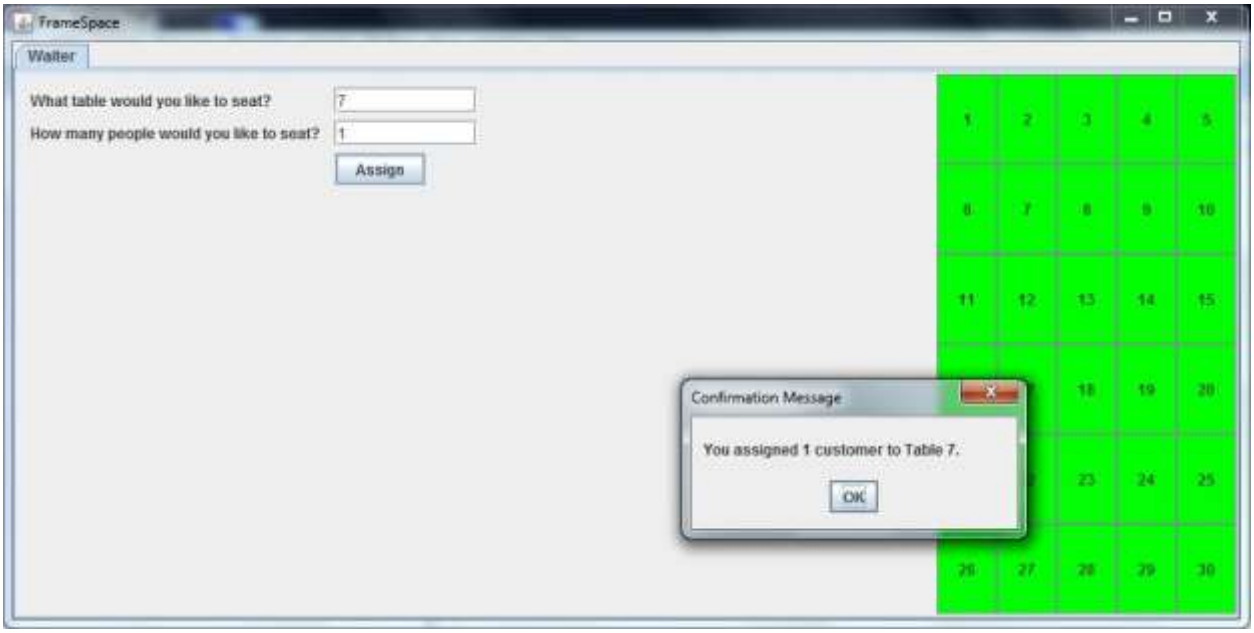


Figure 7 –figure shows the the right message when the values in the text field are correct.

We also tested the JLists in the Table tab to make sure that was receiving an item when you added. We also tested the items in the menu to make sure they would display their descriptions when we highlighted the corresponding item.

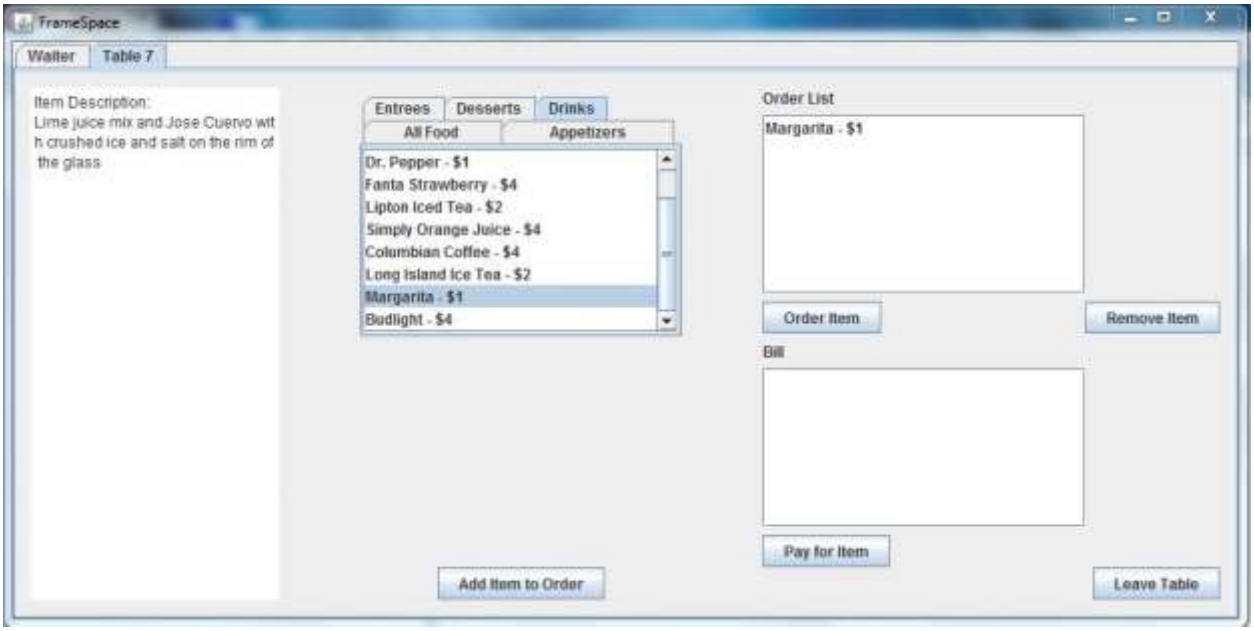


Figure 8 – figure shows the ability to add food to the order list along with item description

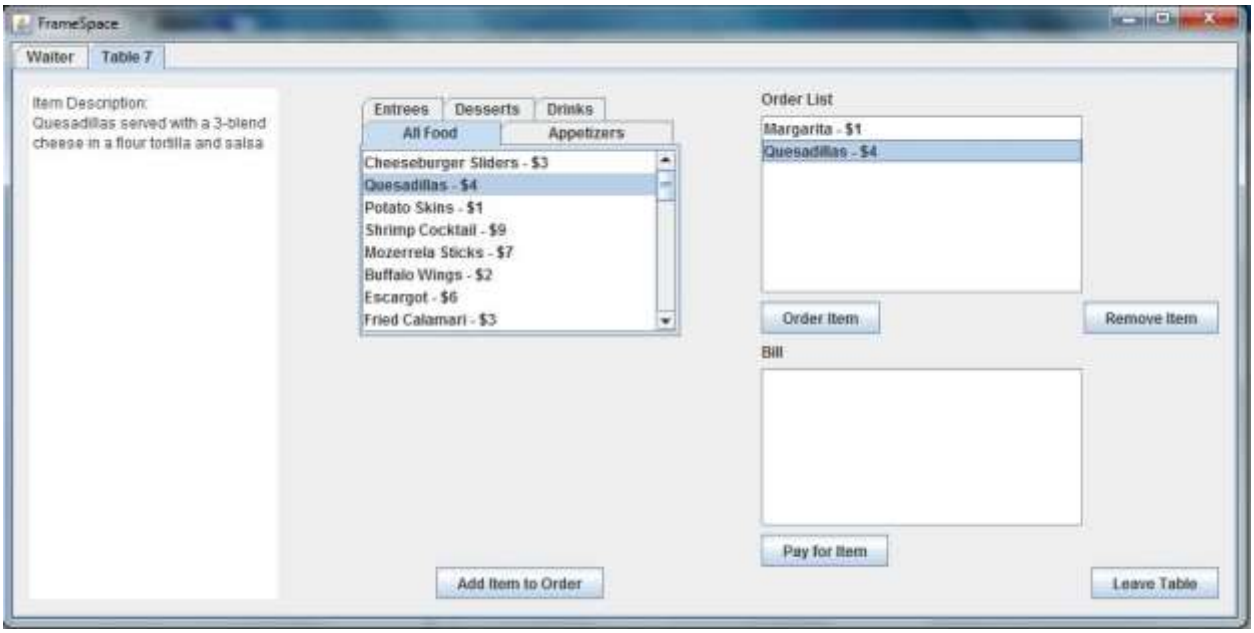


Figure 9 – Figure above show what display looks like before removing an item from the list

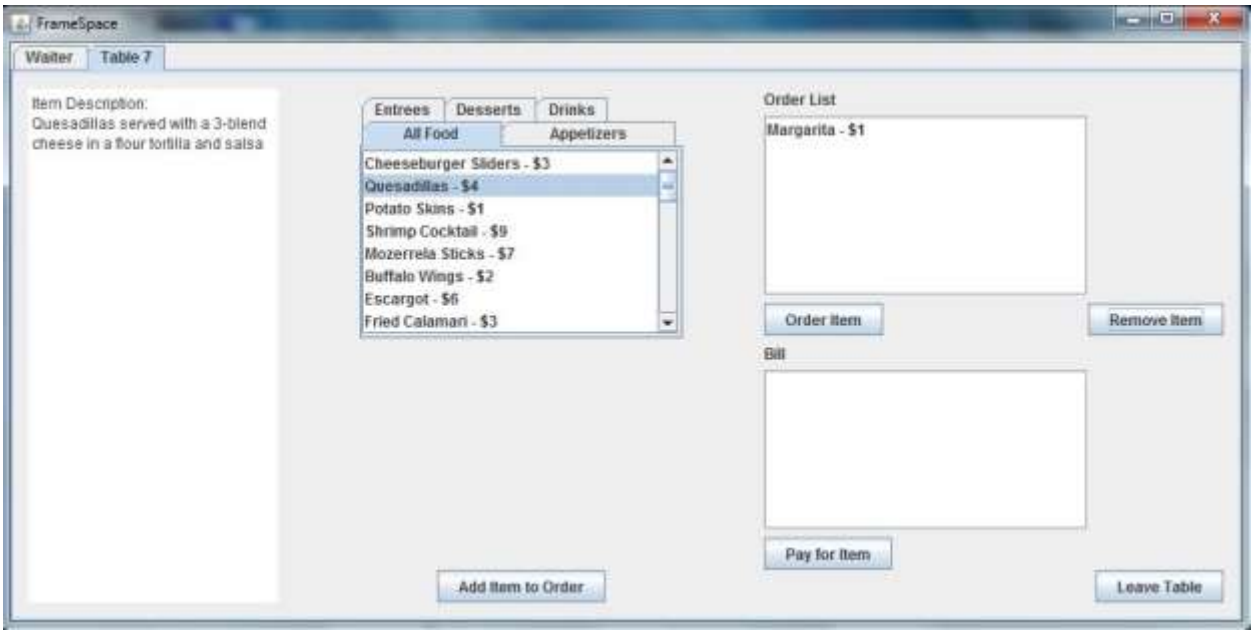


Figure 10 – figure above displays how the GUI looks after item is removed

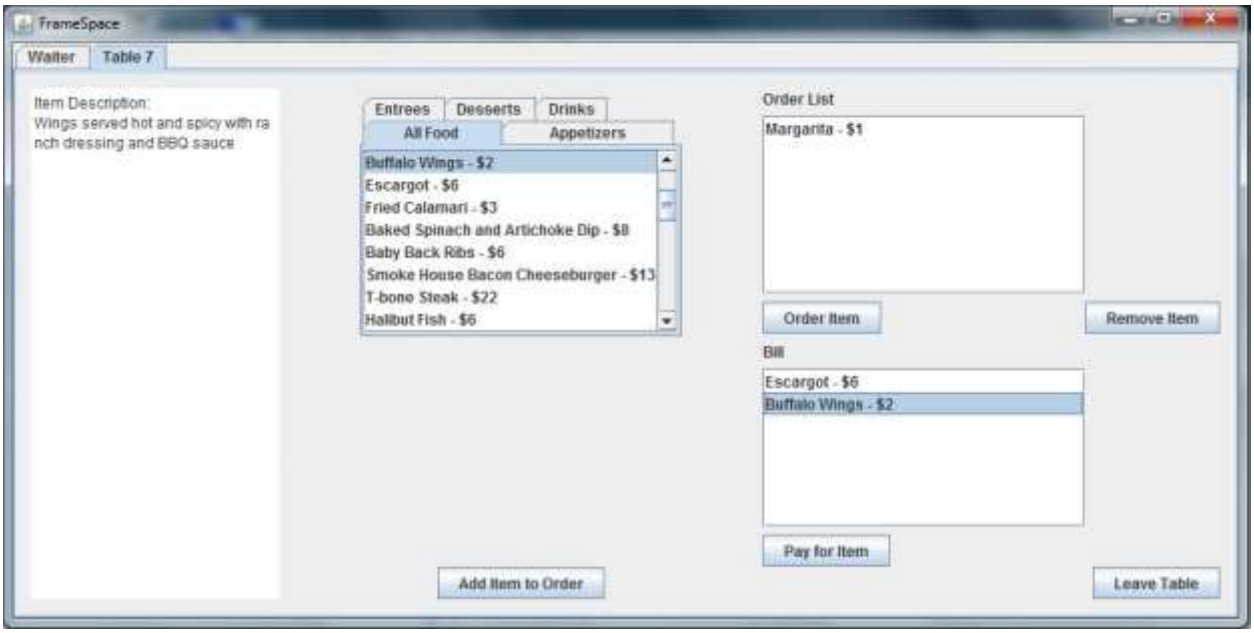


Figure 11- Figure shows what the bill JList looks like before paying for the item.

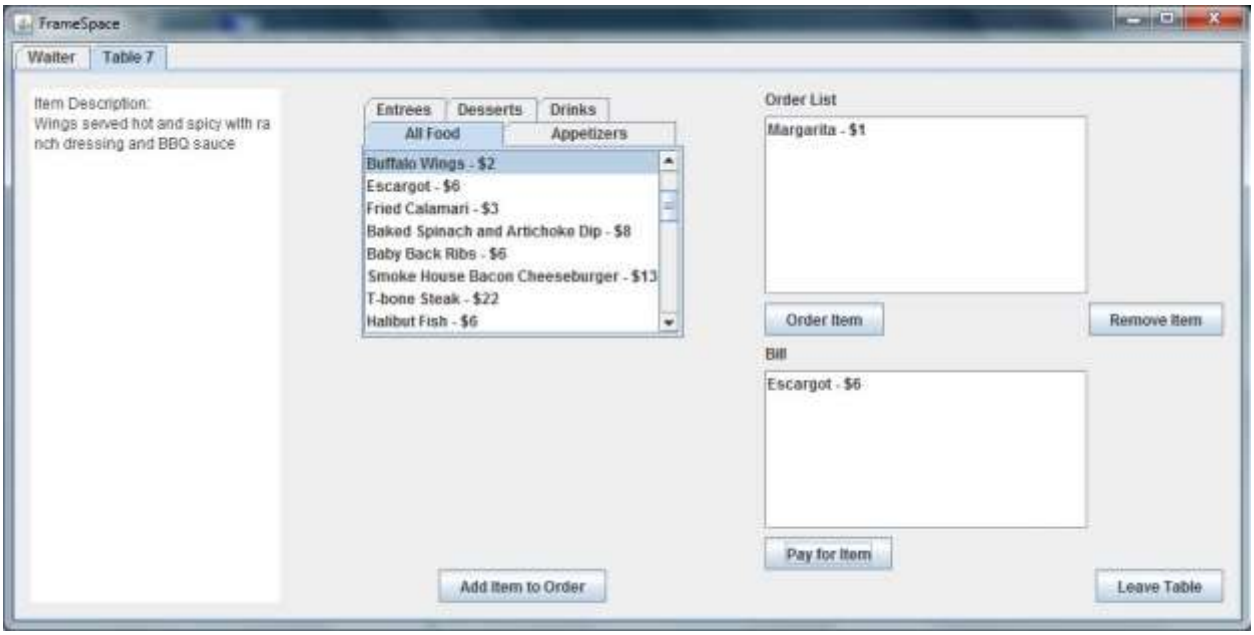


Figure 12 – After the item is paid for

Lastly we tested the capacity of the tables to make sure that it would not let us add more to the specified table and would give us an error message. At this time we also checked to see if the colors for the different tables would change between green, yellow, and red.

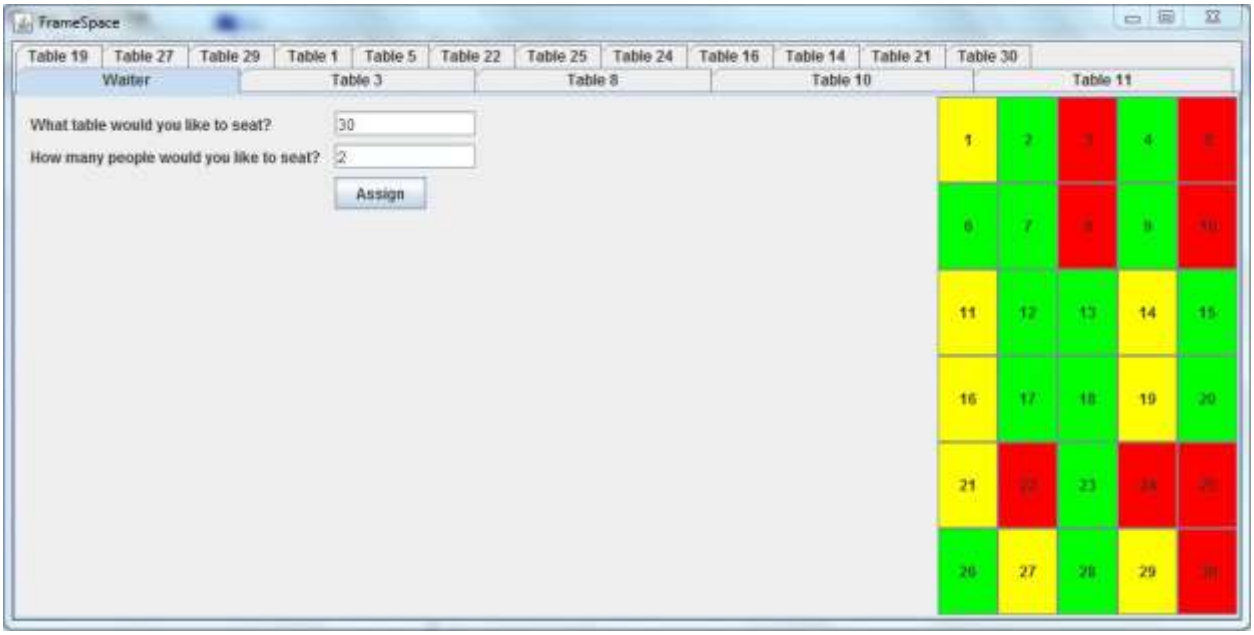


Figure 13- Figure displays the different states of occupancy also shows multiple tabs for each table to add the food.

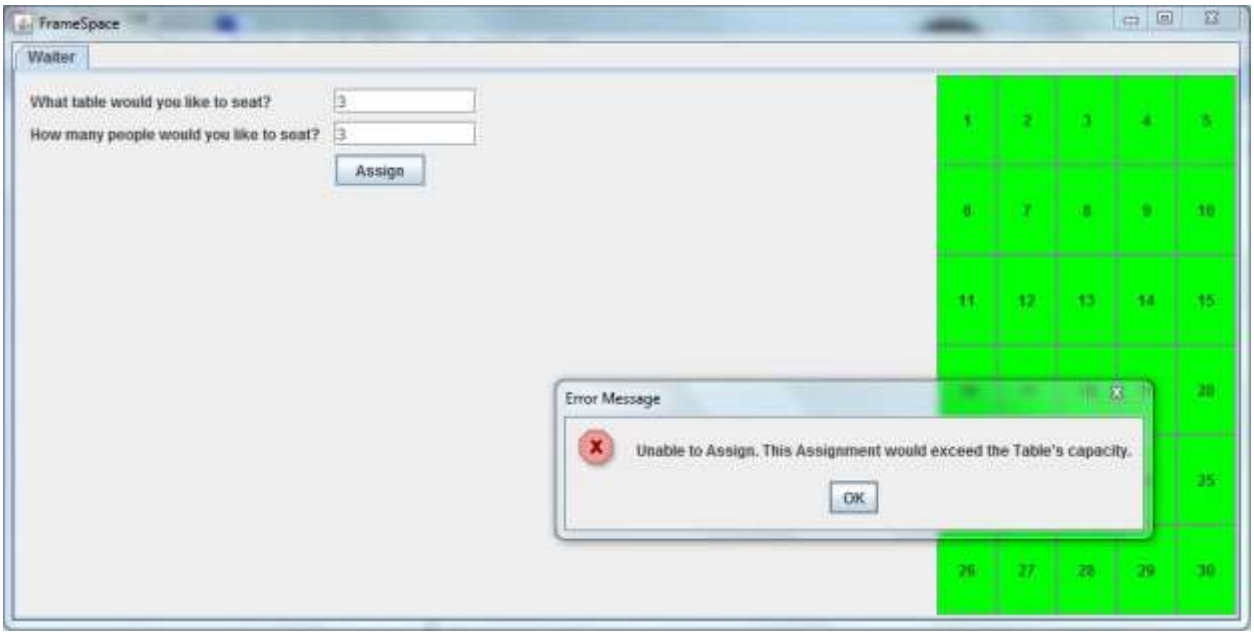


Figure 14 –figure shows what happens when a table exceeds capacity.

Extensibility:

This project has many things that can be added to it to make it even better and more extravagant. Sadly we did not have enough time to create these different extensions.

We can add a money system where people can add in a budget through a JTextField in the Table tab and then when ordering food so they can make sure they are able to pay for the food, and also adds a text field for tips for the waiter. In order to bring this money system to fruition, we would first add a double instance variable to our Customer class, called budget. We would then move over the payOrder method from the Table and place it in the Customer class. This way, a Customer can order his own food and subtract the appropriate amount of money from his own budget. Then, to create the GUI component, in our current java file named AssignmentPanel.java, we would add a JLabel with text asking for a budget, and we would create a JTextField for the user to input the budget (as a float).

We also thought of adding the element of time for the cooking of food and wait time to be seated. Our current understanding of keeping time within a GUI is limited, but once this became clearer to us, we could then pursue it.

Adding a cook/kitchen class that will create the food is also a way we can extend our current restaurant system. First, we would have to create a new class either named cook/kitchen. Since a Table object orders food, we will need a Waiter object to communicate between the newly created cook/kitchen class and the Table, thus reintroducing the Waiter class. The purpose of this class would be to taking time to make items, so that the restaurant system seems more realistic. Again, this concept requires keeping time, and would necessitate more understanding of keeping time within a GUI before we could even map out how to implement the cook/kitchen class.

For the User we also thought of adding a call for waiter button. First, we must create a Boolean instance variable in the Table object , perhaps called flag. Whenever this is set to true, the customers at the table would like to talk to the waiter. In the GUI, the CustomerPanel object contains all of the information pertaining to customers currently seating, and depending

on what side of the screen we want the waiter button to appear (east, west, center) we would add a new JButton to that panel. When pressed, the button would change the newly created instance variable from false to true, to signal the waiter to visit the table. If already true, the button press would do nothing to change the state of the flag.

Lastly we thought it would fun to create a button called “Dine and Dash.” This button would simulate the real-world situation where customers sit down to eat at a restaurant but leave without paying. This action can sometimes result in the customers not paying anything for their meal. Other times, customers are arrested, or they are required to wash dishes for the restaurant. In our program, we will create similar outcomes. For example, we will begin by placing a JButton with the text “Dine and Dash” on the CustomerPanel. When pressed, this would activate a newly created method in the Customer class called dineAndDash. Based on a random number generator, one of three outcomes would result: 1) The customer successfully leaves the restaurant immediately and the customer’s budget is unchanged. 2) The customer’s budget is unchanged but he cannot leave for a specified amount of time (simulating dishwashing). 3) The customer leaves immediately but his budget is a quarter of what it used to be (simulating going to jail).