Building the World out of Objects

EEE320 lab 5

OCdt Aaron Brown
And
OCdt Seth Gillingham

Submitted 25 November 2022

Introduction

This lab was conducted in order to expose us to more UI elements of the project and apply the skills learned in lectures with less structure. This was done by adding in support for bill functionality to the OORMS system. The Bill functionality includes support for preparing, printing, changing and logging bills.

Discussion

Implementing the Bills class

As there will be many methods required to manipulate the bills of a table a *Bills* class was created in the model file. The *Bills* class contains the methods that operate the buttons “One Bill”, “Seperate Bills” and “Combine Bills”. It also has an attribute of type list called “ledger” that records every transaction that happens at the restaurant. The Bills class is utilised in a similar way to view and restaurant. We added a new attribute to the *Controller* class “__init__” method called bill making it required that all the controllers hang off the bill class whenever the controller changes. The class *RestaurantController*’s “__init__” method has a slight exception where if bill is not provided a new instance of *Bills* is created.

Implementing item states

The items stored at each seat at a table are required to have a state in the implementation of the OORMS system. To implement this functionality the *OrderItem* class was modified. The “ordered” attribute was modified to store an int rather than a bool increasing the number of states available for each item. For this version of OORMS the only states required are unordered, ordered, and served, represented by 0, 1 and 2 respectively. Each method in

OrderItem was then modified to reflect the state implementation change. The “mark_as_ordered” method was changed to set the “ordered” attribute to “1”. The “has_been_ordered” method was modified to check if the current state of ordered is “1”, if so return True, else False. The “mark_as_served” method was created to change the state of ordered to served in the same method as “mark_as_ordered”, The “has_been_served” method works in the same way as the “has_been_ordered” method except ensuring the state is equal to “2”. Finally, the “can_be_cancelled” method is modified to check if the state is less than or equal to “1”. This is because if the state is less than or equal to “1” the food has not been served yet. These methods will be used in order to mark the orders as served and eventually reveal the bill buttons.

Marking as served

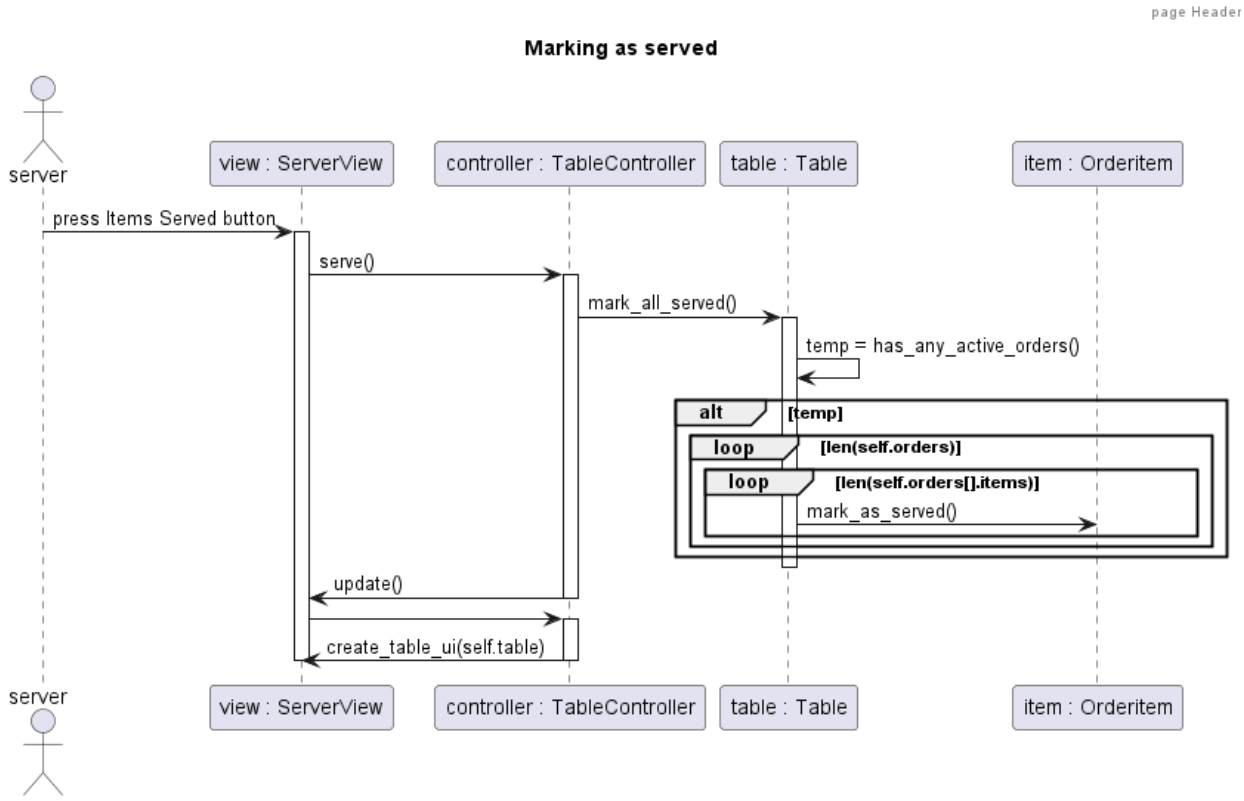

Figure 1: sequence diagram for when the server presses the mark as served button

Use Case for marking items as served

Server has served all items to the customers and want to notify the system

preconditions: The Server is logged into the system and the restaurant view is visible. restaurant view is on Table View.

1. The server selects the button labeled "Items Served".
2. The system will change the state of all items ordered at the table to served

3. The system will remove the button labeled "Items Served" and replace it with three buttons saying "One Bill", "Separate Bills" and "Combine Bills"

Alternates and exceptions:

1. For step 1 the Server could have pressed the button labeled "done" returning them to restaurant view

Figure 2: Use Case for when the server presses the mark as served button

In the OORMS system the items require states such as unordered, ordered and served in order to keep track of where food is in the restaurant. Once a table receives all their food the server will then press the button labelled “Items Served” on the bottom left hand corner of Table View. This button will then trigger the events depicted above in figure 1. The handler for the button will first call the *TableController* method “serve”. This method will then call the method “mark_all_served” in the *table* object. The “mark_all_served” method was placed in the *Table* class because the *Table* class is responsible for all the seats at a table. As the *Table* class is in the method this is where the computations can occur. Within the “mark_all_served method it is first ensured that there are current active orders. If true, a for loop with a nested for loop will go through the orders list in the *Table* class. This will effectively cycle through each item at each seat marking them as served by calling the “mark_as_served” method in the *item* object. After the serve method in *TableController* will call the update method in *ServerView*. Redrawing the server view is necessary because the button labelled “Items Served” should no longer be displayed and replaced with the buttons “One Bill”, “Separate Bills”, and “Combine Bills”.

In this section of the lab we faced difficulties in creating the tests due to lack of confidence with the unittest framework and test-driven development. We were initially going to test if the bill functions could be called prior to all the items states being set to serve; however, due to our lack of in-depth understanding of unitest, a new approach was taken. The initial approach was abandoned because we could not find how to treat an error as a pass. To compensate for our shortcomings we created a test that will ensure that after pressing the “Items Served” button all items ordered at the table have been moved into the served state. This was much simpler as “assertEqual” could be used. Similar Testing methods are used in the following sections.

One bill

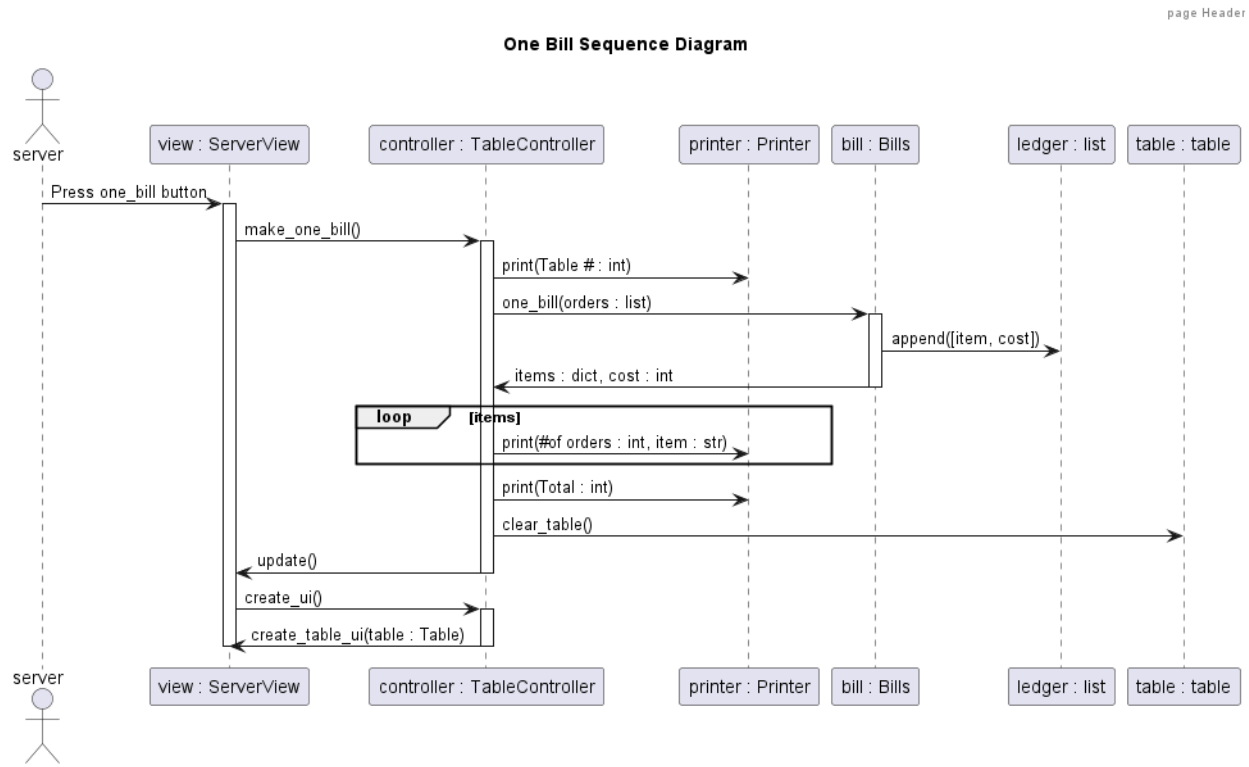

Figure 3: sequence diagram for when the server presses the One Bill button

Use Case for One Bill:

Server wants to create a bill with all items at the table with a single cost

***preconditions*: The Server is logged into the system and the restaurant view is visible. restaurant view is on Table View.**

1. The server selects the button labeled "One Bill".
2. The system adds the table number, all ordered items at the table and the total cost on to the printer tape.

Alternates and exceptions:

1. For step 1 if not all items at the table are served the button labeled "One Bill" should not be visible

Figure 4: Use Case for when the server presses the One Bill button

Once all the ordered items have been served and the table wants to pay altogether, the server will press the “One Bill” button. This button will then trigger the sequence of events depicted above in figure 3. This feature was implemented initially to assist with testing of our system as it is much simpler to implement than separate bills. We left it implemented as it could be a useful feature for a legitimate OORMS system. The handler for the button will first call “make_one_bill” in the *TableController* class initiating the printing. The controller then will print the table number

as a header to the bill. The information for the bill is then generated using the “one_bill” method in the Bills class. This will return the items ordered by the table in the format of a dictionary and an integer holding the total cost. The dictionary contains the items ordered such that the keys are the food and the item is another list. The list in the dictionary’s zero element is the price of the item and the second element is the number of times it was ordered. This method of data structuring was decided upon as it separates the two major pieces of data required (items to print and total cost) into clean easy to use variables. Before “one_bill” returns, the information is added to the log in the form of a list where the zero element is the dictionary and the first element is the total cost. Once the data has been returned from “one_bill” it is printed to the printer tape using a for loop. The table is then cleared removing all the orders, preparing the table for the next customers and the view is updated. The view is updated as the colouring of the seats and the shown buttons on table view will now change.

Minimal difficulties were faced when creating this feature other than the conception of appropriate data types and structures. This feature was tested by creating orders at a table and comparing the output of “one_bill” to premade results.

Separate bills

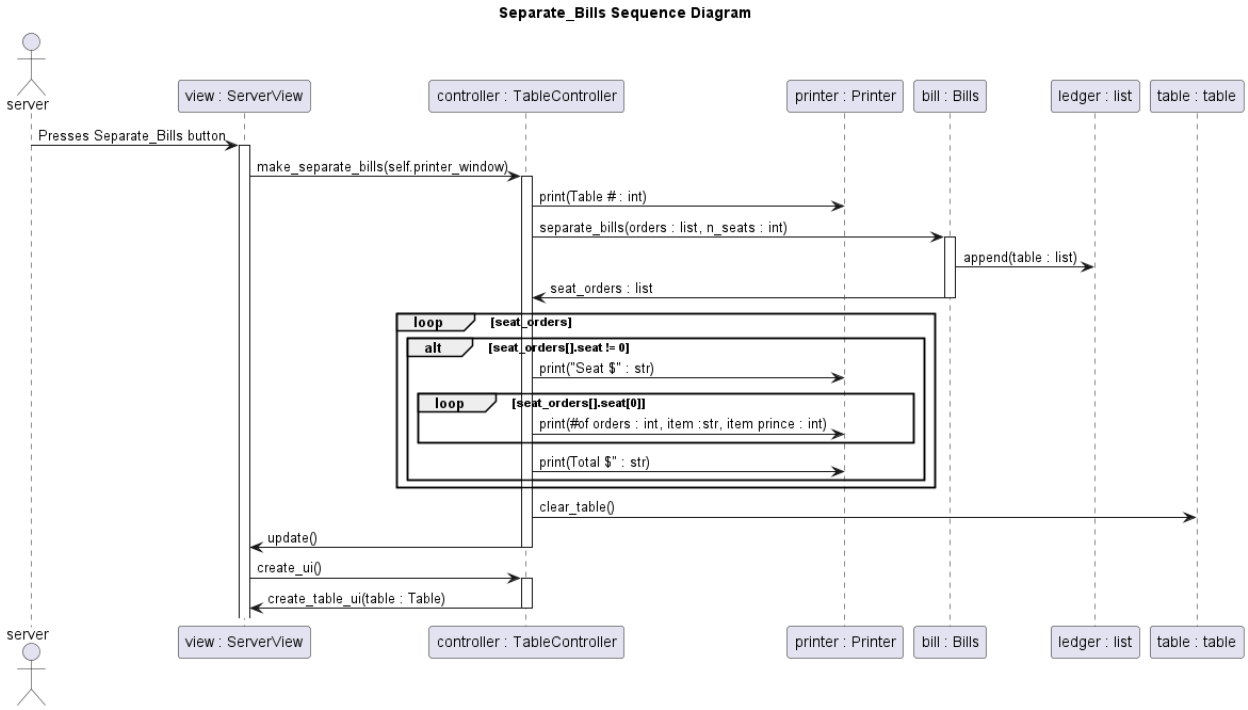

Figure 5: Sequence Diagrams for Separate Bills button

Use Case for Separate Bills:

Server wants to create individual bills for each seat that ordered at the table

preconditions: The Server is logged into the system and the restaurant view is visible. restaurant view is on Table View.

1. The server selects the button labeled "Separate Bills".
2. The system adds the table number, and for each seat the seat number, all ordered items at the seat and the total cost for the seat on to the printer tape.

Alternates and exceptions:

1. For step 1 if not all items at the table are served the button labeled "Separate Bills" should not be visible

Figure 6: Use Case for Separate Bills button

Rather than pressing the “One Bill” button, the server has the option to press the “Separate Bills” button. This button is used when all the table items have been served and the customers want to pay individually for their seats. This can also be used to pay for some separate bills and some combined when used in conjunction with the “Combine Bills” button. When pressed the button will trigger the sequence of events depicted above in figure 5. The button’s handler will first call the method “make_separate_bills” in the *TableController* class. This will call the printer to print the table number as a header to the bill then prepare the data of each bill by calling “separate_bills” in the class *Bills*. The method “separate_bills” collects each seat's orders and returns it in the form of a list. The list consists of representations of seats in the form of a “0” if there are no ordered items or another list if there are ordered items at the working seat. The version of a seat representation that is a list is the same as the return from one_bill however is for a seat rather than the whole table. After a sequence of for loops is used to print out the seat number, food items order and the total for each seat using the *printer* classes print method. The list returned from separate_bills is then added to the log and the view is updated like in one_bill.

This section of the lab caused significant difficulties due to the complexity of the data structure used. The data structure used was utilised due to its elegance however it is also very complicated. After drawing it out on paper further it became easier to grasp leading to the creation of the nested for loop. The feature was tested in the same manner as one bill where a premade result is compared to the output of “seperate_bills”.

Combine bills

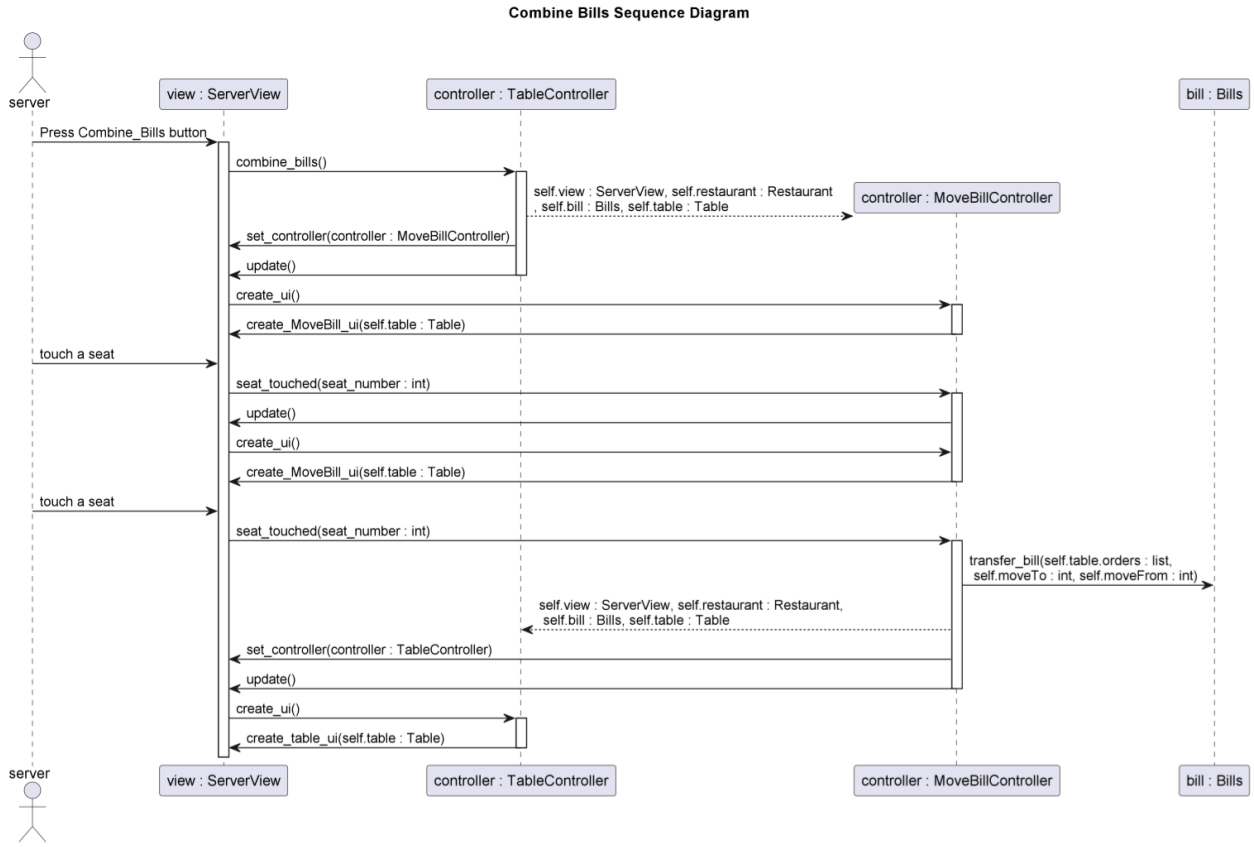

Figure 7 : Sequence Diagrams for Combine Bills button

Use Case for Combining Bills

Server wants to move the cost of one seat to the cost of another seat

preconditions: The Server is logged into the system and the restaurant view is visible. restaurant view is on Table View.

1. The server selects the button labeled "Combine Bills".
2. The system then displays the table
3. The server will select the seat that they wish to move a bill to
4. The system will display the colour of the clicked seat as green
5. The server will select the seat that they wish to move the bill from
6. The system will move the bill and display Table View

Alternates and exceptions:

1. For step 1 if not all items at the table are served the button labeled "Combine Bills" should not be visible
2. For step 3 through 5 the Server could instead press the button labeled "cancel" and it will return to Table View

Figure 8: Use Case for Combine Bills button

In order to make combining bills user friendly, the button will change the controller in ServeView to a new controller class called *MoveBillController*. It is responsible for creating a new screen where the server is presented with a zoom in of the table similar to Table View. The screen will also have a button labelled “cancel” in the bottom left that will bring the server back to Table View. The *MoveBillController* inherits from the *Controller* class and was created to have a unique page for when combining seats bills. The *MoveBillController* class is modelled after the *TableController* class sharing many traits such as the attribute table and the seat_touched method. However, the *MoveBillController* class has two additional attributes called “moveTo” and “moveFrom”, used to store what seats are combining their bills.

The sequence described in figure 7 shows the events that will happen if the server wants to move an order. First the server will press the button in the bottom left of Table View labelled “Combine Bills”. The button will then change the controller to *MoveBillController*, changing the display. The Server will then touch the seat that will be receiving the bill. The seat will turn green. The Server will then touch the seat that will be sending the bill. The first seat touched will then have the second seat's orders appended to its own and the second seat will have its orders cleared. After The controller will be handed back to *TableController* reverting the display to Table View.

Due to the fact that the methods in *MoveBillController* are very similar to the ones used in *TableController* minimal issues were faced. To ensure proper function of this feature three tests were created. The first test creates orders on multiple seats at a table then moves one order. After moving the order “separate bills” is called in the *Bill* class and the output is compared to a premade dictionary. The second test is the same as the first other than “one_bill” is called rather than “separate bills”. The final test ensures combining bills with a seat that has no orders works by comparing the output of “separate bills” to a premade dictionary.

Conclusions

In this lab we learned many lessons about how to manage and plan for a larger project. Throughout our university career most programming labs were relatively small programs which didn’t force strict organisation. This lab exposed us to new changes as without good planning it makes it very changing to collaborate on projects that have many components. All skills learned in the lectures of the course were utilised in order to complete this lab.