OORMS V0.1: Taking Orders
EEE320 lab 3

NCdt Murray, 30711
OCdt Jiang, 29761
Submitted October 2nd 2024

Introduction

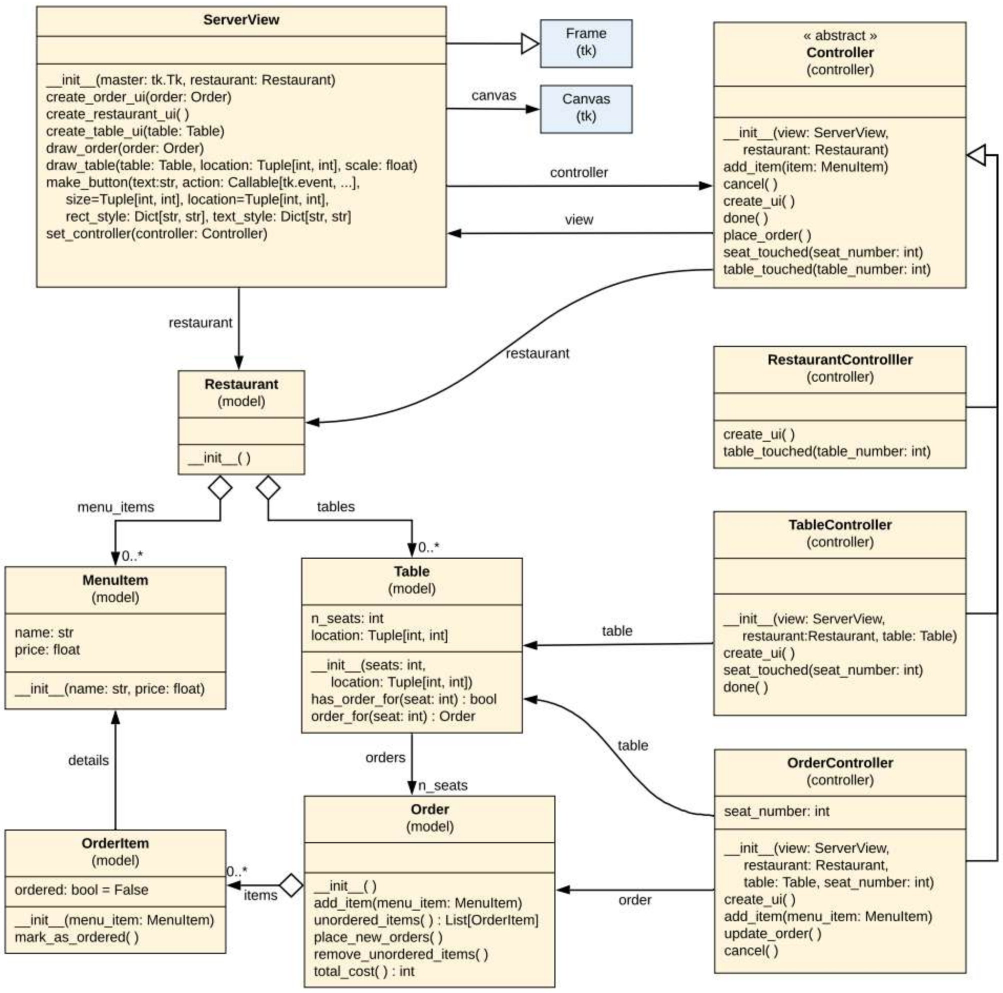
The aim of this lab was to familiarize ourselves with object oriented programming by implementing a predesigned restaurant management software in python called Object Oriented Restaurant Management System (OORMS). Sequence diagrams were provided describing the operation of a ‘take order’ use case. Class diagrams were provided describing the required classes and their attributes. During the lab, these diagrams were implemented in python code.

Discussion

Object Oriented Programming (OOP), as a programming paradigm, excels at representing real world objects and interactions. Making it ideal to handle restaurant orders.

UML Class Diagram

In UML, the methods, attributes, and interactions of classes are described by a class diagram. Below is the class diagram for the OORMS program.


ServerView
__init__(master: tk.Tk, restaurant: Restaurant)
create_order_ui(order: Order)
create_restaurant_ui( )
create_table_ui(table: Table)
draw_order(order: Order)
draw_table(table: Table, location: Tuple[int, int], scale: float)
make_button(text:str, action: Callable[tk.event, ...],
size=Tuple[int, int], location=Tuple[int, int],
rect_style: Dict[str, str], text_style: Dict[str, str]
set_controller(controller: Controller)
Frame (tk)
canvas
Canvas (tk)
controller
view
« abstract »
Controller
(controller)
__init__(view: ServerView,
restaurant: Restaurant)
add_item(item: MenuItem)
cancel( )
create_ui( )
done( )
place_order( )
seat_touched(seat_number: int)
table_touched(table_number: int)
restaurant
restaurant
Restaurant
(model)
__init__( )
RestaurantControlller
(controller)
create_ui( )
table_touched(table_number: int)
menu_items
tables
0..*
0..*
MenuItem
(model)
name: str
price: float
__init__(name: str, price: float)
Table
(model)
n_seats: int
location: Tuple[int, int]
__init__(seats: int,
location: Tuple[int, int])
has_order_for(seat: int) : bool
order_for(seat: int) : Order
TableController
(controller)
__init__(view: ServerView,
restaurant:Restaurant, table: Table)
create_ui( )
seat_touched(seat_number: int)
done( )
table
table
details
orders
n_seats
OrderItem
(model)
ordered: bool = False
__init__(menu_item: MenuItem)
mark_as_ordered( )
0..*
items
Order
(model)
__init__( )
add_item(menu_item: MenuItem)
unordered_items( ) : List[OrderItem]
place_new_orders( )
remove_unordered_items( )
total_cost( ) : int
order
OrderController
(controller)
seat_number: int
__init__(view: ServerView,
restaurant: Restaurant,
table: Table, seat_number: int)
create_ui( )
add_item(menu_item: MenuItem)
update_order( )
cancel( )

- ServerView

The ServerView class controls the interactions between all controllers, the restaurant model, the tk.frame, and the tk.canvas. The white arrow from serverview to frame represents an inheritance relationship, wherein ServerView instantiates frame objects which can exist independently from ServerView. The ServerView passes all restaurant information to Canvas and Frame to generate UI, and all user instructions to the controllers.

create_restaurant ui is called in the constructor of ServerView. Create_table_ui and *create_order_ui* are called from controllers when a table or seat is touched. They use tk.inter to create the UI.

Set_controller sets the active controller.

- Controller <<abstract>>

The abstract controller class, and its subclasses, serve as tools to interact with Restaurant, Table, and Order items. Controllers are required to direct control flow to each required model item.

When a table is touched from the restaurant view, a table and a table controller is instantiated. When a seat is touched from the table view, an order and an order controller is instantiated.

Each controller also has a create ui method to communicate ui information to the tk.frame through the view object.

- Restaurant(Controller)

Table_touched occurs when tk.frame detects that the user has selected a table. It creates a table object and a table controller. It then passes control to the table controller through *view.set_controller*.

- TableController(Controller)

Seat_touched occurs when tk.frame detects that the user has selected a seat. It creates an order object and an order controller. It then sets the order controller as the active controller through *view.set_controller*.

The table controller has the *done* method to indicate when the table is complete taking its order, and pass control back to the restaurant controller.

- OrderController(Controller)

Add_items adds an item to the order through *order.add_item* and updates the UI to display the change.

The order controller has the *cancel* and *update_order* methods to return control to the table controller and “lock in” order information, or cancel it when appropriate.

- Restaurant

The Restaurant object has aggregate relationships with two objects: MenuItem and Table. In fact, when the restaurant object is initialized, it automatically creates a list of Tables and a list of MenuItems. Restaurant objects have no other methods other than its **init** method.

- Order

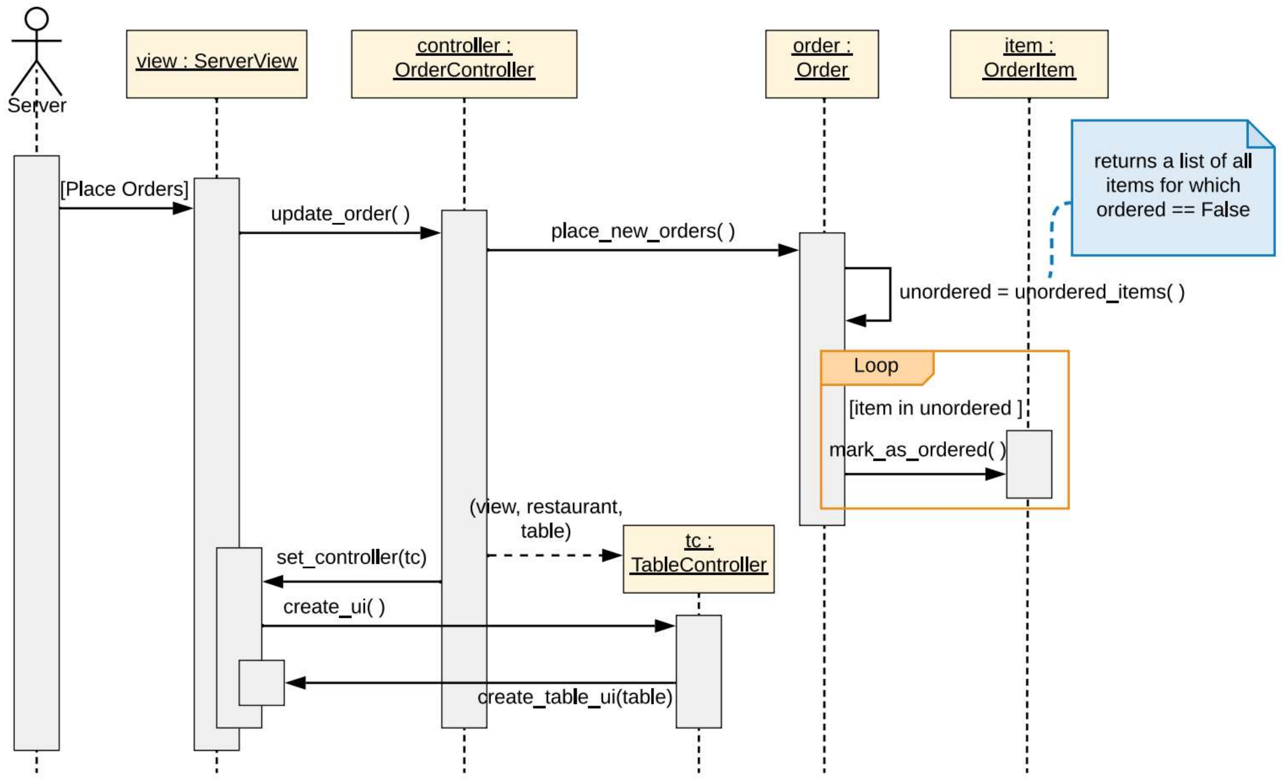
Contains by aggregation a list of OrderItem objects that starts out empty. Items that have been selected, but not yet confirmed are added to the *unordered_items* list. When the server selects Place Orders, all items from *unordered_items* are removed. If the server selects cancel, all unconfirmed items are deleted with *remove_unordered_items. Total_cost* reports the total cost of all items in the order, confirmed or not.

- Table

Tables are objects that have an aggregate relationship with restaurants: restaurants typically have 0 or more tables. Table objects are therefore created when the restaurant object is initialized. Table objects also interact with order objects because tables have seats and each seat can have an order. We can refer to order objects and interact with them by indexing into the seat number of a table.

UML Sequence Diagram

In OOP, a UML sequence diagram describes the interactions of objects. To demonstrate the design process and our interpretation through our code, we will walk through a sequence diagram (Place Orders) that interacts with many objects.



Each beige square indicates a participant in object:Class format. The timeline of the sequence starts from top to bottom. The arrow that points to a participant often represents a method. The participant that the arrow points from is the one that calls the method and the participant that the arrow points to is the next participant that does an action following the method call. The participant enters the sequence diagram when the dotted line transitions to the solid line and leaves the sequence when the solid line transitions to the dotted line.

The diagram starts with the actor, who is a server in our scenario. When we run the program, the server is the person interacting with the GUI. In this sequence diagram, the server wants to place all the orders that are currently unordered in a particular seat. The server initiates this sequence by clicking on the 'Place Orders' button, assuming that there are a few unordered items in the order. The creation of the button is represented by the following line of code in oorms.py:

```
self.make_button('Place Orders', lambda event: self.controller.update_order())
```

By clicking on this button in the GUI, ServerView calls *self.controller.update_order().*Since the button is found in the Order UI, self.controller refers to order_controller which calls the *update_order()* method.

We need to understand that this sequence diagram has two parts: the core functionality and the UI.

```
def update_order(self):

   self.order.place_new_orders()

   self.view.set_controller(TableController(self.view, self.restaurant,
self.table))
```

order_controller.update_order() simply calls the *place_new_orders()* method from its *self.order* attribute.

Here's the code snippet for the *place_new_orders()* method:

```
def place_new_orders(self):

   for item in self.unordered_items():

       item.mark_as_ordered()
```

Every Order object has a list of unordered items as an attribute which starts out empty. When the server clicks on items in the Order UI, it appends those items to the *unordered_items* list. Unordered items are OrderItem objects, so they have the *ordered* attribute which is a boolean value that starts out false. When *place_new_orders()* is called, we simply iterate through the unordered items and we change the *ordered* attribute of each item to true.

The core functionality of this sequence diagram is done, we now want to reflect it in the UI. We need to understand that before we kick off the sequence diagram, we are in the Order UI, after we click on the Place Orders button, we have to go back to the Table UI. Therefore, as soon as the *place_new_orders()* method is called, OrderController initializes a TableController object and sets the controller of the view to this object. This is done by the following line of code.

```
self.view.set_controller(TableController(self.view, self.restaurant,
self.table))
```

When the set_controller method is called, it automatically creates the UI of the controller, which is a table in this case.

```
def set_controller(self, controller):

   self.controller = controller

   self.controller.create_ui()
```

Since *create_ui()* is an abstract method, the controller it was called from overrides the method.

```
def create_ui(self):

   self.view.create_table_ui(self.table)
```

Now the GUI should exit the Order UI and go to the Table UI.

Encountered problems and solution

Problem 1:

When we ran the program, all seats were colored green by default.

Explanation:

The coloring of each seat is determined by the following line of code in *draw_table*

```
style = FULL_SEAT_STYLE if table.has_order_for(ix) else EMPTY_SEAT_STYLE
```

Where the method *table.has_order_for(ix)* is intended to return whether or not a seat has an order:

```
def has_order_for(self, seat_number):
   return self.orders[seat_number]
```

However, an order always exists under a table or a seat, therefore the method will return as true even if the order is empty.

Solution:
The method needs to instead look to see if the order object has an item:

```
def has_order_for(self, seat_number):
   return self.orders[seat_number].items
```

Problem 2:

Our program would fail *add_item()* method test cases concerning the *create_ui()* function and have very weird UI interaction when we ran the program and tried to add items to our order.

Explanation:

Every single sequence diagram, which are actions that the server takes, interacts with the UI in some way. For most actions, the UI changes after the action is done. For example, if we are in the Order UI and we cancel the order, we go back to Table UI. We assumed that all actions would result in a UI change so we took a shortcut and we called the *create_ui()* function in the same way for every situation, which is usually called from another controller object that we instantiated to create a different UI.

Erroneous code:

```
def add_item(self, item):
   self.order.add_item(item)
   self.view.set_controller(TableController(self.view,self.restaurant,
   self.table)
```

However, when we add items to our order, we stay in the Order UI.

Solution:
The actual solution is a pretty simple fix, but it requires us to have an understanding of how the UI changes as we take actions. We simply had the *add_item()* method to call the *create_ui()* method itself through the OrderController instead of calling the method by initializing another controller object.

```
def add_item(self, item):
   self.order.add_item(item)
   self.create_ui()
```

Conclusion

This section was significant to our understanding of OO development. By following pre-existing. UML diagrams, we learned interfacing with event driven programming. This was also our first exposure to event objects.
This lab had most of the design work completed for us. The next steps would be for us to design our own OO program and create our own UML class and sequence diagrams.

It was also the first time we worked with sequence diagrams, so it was really fun to read the sequence diagram and implement the diagrams' main ideas into Python code. It allowed us to get a full understanding of the role of each participant and how they interact with each other.

References

1. Capt Sullivan's PowerPoint slides and sequence diagrams, Accessed from moodle 19 September 2024.
2. Martin Fowler, *UML Distilled Third Edition: A Brief Guide to the Standard Object Modeling Language*, Addison-Wesley, 2004. ISBN 0-321-19368-7.