Ninja Fast Kitchen System 3000

By: Louis Lu, Ethan Tuttle, John Foster, Shawn George

Table of Contents:

About

In November of 2020th the world shut down as Covid forced countries to halt normal operations. After a tough year of isolation America began its movement to reopen stores and shops nationwide. In the following weeks restaurants across the nation suddenly found themselves with an influx of customers that they simply were not prepared to handle. For family restaurants this was an unprecedented event. As a result smaller businesses were left with no option besides turning away customers at the door and phone or letting them know that they would have to wait for hours or more for their food. However, this was also an opportunity that smaller businesses were not ready to

give up so easily. Therefore a small frenzy ensued as restaurants scrambled to get orders out of the door as fast as possible while desperately trying to keep up with the increasing demand. Our proposal to deal with the sudden flood of orders is a Kitchen Display System that provides not only the traditional information including orders, and wait time but also suggests a priority queue in order to sort which order should be finished first.

Backend

In order to implement this display system we begin by creating several key classes. Of note are: ActiveOrders, Customer, Menu, MenuItem, OrderComparator, and OrderItem. On a higher level these classes represent exactly the items they are named after in our display system. To begin, let us describe the MenuItem class.

The MenuItem is a class which effectively holds three key fields: name, timeToMake, and category. The name is the name of the menu item, the timeToMake is the estimated time it takes to actually make this menu item and the category is the category this menu item falls under in the restaurant's overall menu.

This MenuItem is then utilized in the OrderItem class. The OrderItem is a class which effectively represents an order that a client may place for a restaurant. This class contains two key fields: order, and timeToMake. Here the order is a structure which holds all of the menu items a particular order would contain. Furthermore, time to make would contain the overall time necessary to create the entire order based on the estimated time to make when creating each individual menu item.

This OrderItem class is then utilized by the Customer class. The Customer is a class which effectively represents a client ordering food from a restaurant. The Customer object contains a number of values which are necessary for deciding the order

of a Customer object in the priority queue. These variables are allOrders, singletonOrder, timeToMake, name, ticksInQueue, orderPlaced, and timePlaced. The singletonOrder acts as a single Order object which is populated and then added to the allOrders structure which contains all the orders a customer may have placed. At a high level this allows for the extension of customers placing multiple orders for multiple individuals including themselves. There are also name and timeToMake fields which contain the name of the customer and the time it would take for a customer's total order to be finished. Along with the ticksInQueue values these values determine the placement of an order within the priority queue. Finally timePlaced contains the initial time an order is placed which will track how long a particular customer has been waiting in the queue.

Each of these Customer objects will then be utilized by ActiveOrders. ActiveOrders is a class which effectively contains the active orders a restaurant is waiting to complete. Each ActiveOrder object contains a structure which contains Customer objects. Here each Customer object abstractly represents an order that the restaurant is waiting to complete. Furthermore each Customer object is sorted by a custom comparator called OrderComparator which is how our application decides exactly where a customer would go in the priority queue which concretely is the order in which the orders should be completed by the restaurant.

Therefore our final key class for this theory of operations is OrderComparator which underlies the logic used to create the priority queue in which our graphical user interface displays the active orders placed by customers. This OrderComparator works by taking two customers and determining which customer's order has a shorter time to make. The shorter time to make wins out in the priority queue. However, if the time to

make is the same then the customer order is arbitrarily decided based on which customer placed their order first. Furthermore there is an overall tick count that orders in the queue receive. If an order has been in the queue long enough then this tick acts to boost the priority of a given order such that a customer is not waiting in the queue forever despite having a longer estimated time to make.

Outside of this connection between classes is the Menu class. The Menu abstractly represents the menu that a restaurant would have and offer to its clientele. The Menu contains a menu which is a structure that maps each category to another structure which maps each menu item name to an actual MenuItem object. In this way the menu is hierarchical such that all menu items fall under a category and all categories cannot be duplicates. It is also then ensured that menu item names cannot be duplicates and must be unique as well. This structure is then utilized by the different graphical user interfaces in order to allow the restaurants to add items to the menu, remove items from the menu, and alter items in the menu. Furthermore customers will then be able to choose from this menu in order to place orders which can be sent to ActiveOrders and then displayed in another graphical user interface.

Socket Connection

Our application uses socket connections to communicate between the Customer application and the Restaurant application. In an ideal world, the restaurant could have a persistent IP to connect to, or a domain name like “google.com” that a Customer application could connect to. Because we don’t have that, each Customer must enter the IP of the machine running the Restaurant application that they would like to connect to.

The Restaurant application creates a thread on startup that waits for connections, and on connection moves that connection to a new thread to communicate with it. This

way it is always looking for new connections and not messing with the actual GUI.cOnce connected, the Restaurant application will send the Customer the menu to be displayed. This is just done by converting all stored items into a string format and sending it. This goes until all items have been sent, and then a line saying “End of Menu”, signals to the Customer application that the whole menu has been sent.

The Customer application also creates a new Thread to parse the incoming data on the socket connection. It then uses a Semaphore to block the order chooser screen from being displayed on the main thread until the menu has been loaded in the socket’s thread. It attempts to establish this connection after the user has submitted their name and IP and immediately locks the semaphore. Once the “End of Menu” line is read by the socket thread the semaphore is released and the main thread creates the order chooser screen. The socket connection is then used by the Customer application to send the customer’s order. This is sent as a single line, starting with the time the order is placed (for determining how long they have been waiting the Restaurant application), their name, and all the names of the items that they order. These are all delimited by semicolons. The connection stays open until the Customer application is closed.

Order Display GUI

This view, called the Active Order Display in the Restaurant views menu bar, shows all orders that are currently in the queue. The queue is just an instance of an ActiveOrders object, and this view just displays items in that queue.

The display is a border layout, with the center being a panel of DisplayItem objects. This panel is a box layout with a horizontal scroll, so once enough items are added to the queue you scroll to see the ones further down the queue. This prevents the layout from squeezing all of the DisplayItems together.

The DisplayItems are a representation of an order, with the name of the customer, the time they have been waiting, and then every item in their order. Given a black border it makes it seem like a ticket on a chef's line of order tickets. The swing components themselves are a box layout with a vertical scroll, so that way long orders just add a scroll bar to move down the order instead of messing with the sizing. There is also a button at the bottom to remove the whole order from the queue and display.

There is also a button at the bottom of the whole display called "Refresh" that refreshes the display, but this is only useful if you have been off the display for an extended time. While you're on the display it auto-updates the display with new orders as they come in. So it can largely be ignored if a restaurant is paying attention to the active orders display for an extended time.

Orders are also sorted as they come in, with shorter items coming since we learned in Operating Systems that a shortest job first algorithm tends to have a higher uptime and better utilization. Orders can obviously be done in any order as they are removed by the restaurant manually, but this process helps suggest a method to complete orders.

Menu Creation GUI

Our application has a restaurant interface where a restaurant user is able to input the categories for their menu and the specific menu items in those categories. The interface is essentially a large panel that the restaurant frame is set to. It encloses other panels within it to provide the interface for adding categories and eventually adding menu items when the categories are added.

The entire panel is a border layout so in the north section is the panel with a flow layout where you can enter a category in a JTextField and click the Add Category button to add the category if the category is valid. Below that panel is a panel with a BoxLayout aligning along the X axis that encloses other categories and their respective menu items when you enter them. Within this panel is a panel with a border layout that encloses a specific category and their items. It holds a panel that has the category label, and button to add a new menu item to that category and another panel with a box layout aligned along the Y axis to hold the actual menu items of that category. As categories are added panels that hold actual categories are added to the right.

The Menu Creation GUI functionality though seeming simple was tough to implement to prevent certain complications in a menu. For our implementation we had to prevent the user from adding duplicate categories and duplicate menu items within a category. To first add a category a restaurant user would enter a unique category value and that would add a panel with the category and the ability to add menu items under that category. If you were to try and add a category that is already there you would get an error that the category is already in that menu. When a category is created you are able to add menu items below it using the Add Menu Item button. Here a dialog box would show up specifying the menu item name with the minutes and seconds a menu item would take to cook. This again would not add a menu item that has the same name. There was also the possibility of changing the name of the category as long as it was distinct from other categories in the menu. The same functionality was available for the menu items as well. The toughness to implement this functionality came in the form of being able to remove items so they could be added again or when we had to replace items. We had to make sure if you were changing a menu item you could keep the same

name and change the time. There were a lot of edge cases but using the underlying map representation for the menu really helped with removal and replacement of values. In the end the checking of these cases made the Menu Creation GUI very robust and able to account for many values.

Customer GUI

The way our application receives orders is through a separate GUI distributed to the customers. When customers run their jar file they will be prompted to enter a customer name, as well as an IP address which is used to establish the socket connection. Upon entering valid values for name and IP address, the GUI will load. The GUI presents the customer with buttons to place order, as well as a live receipt to view their order. Customers can also swap between different categories to order from as well as interact with a back button to return back to the initial category screen. Once a customer has finished ordering all their items, they can choose to place the order. At any time during this process, customers can clear the order which clears the entire receipt. The GUI also blocks customers from placing orders that contain no items, as well as clearing orders that do not contain items. Once a customer places an order it logs it using the MenuReader class, and sends it to the CustomerGUI class over the socket. The order is then added to the active order display, and displayed in the restaurant GUI. Some issues that came about when developing the customer GUI, were integrating the delete order button as well as order categories. The delete order button was a challenge to format properly with the orders as they were added to the receipt. The solution involved using a GridLayout to display orders on JPanels, as well as adding corresponding JButtons with action listeners hooked up to delete the corresponding

panel which contained an order. Another issue involved adding the categories to the GUI, the way we went about this was to first display all the categories present on the menu, when a customer clicked on a category, a function would wipe the panel clean of all buttons, and run through new logic spawning in buttons containing the values of that map given the category string, which in this case where individual menu items. After this function was called, a new button would appear which was labeled <--back. When this button is clicked, the panel is wiped clean of all buttons again and returned to its original state.

Development Process

The development process for this project took a very orderly manner through which we took things from a high abstract level down to a very concrete level. At first we discussed the aspects of the restaurant display system that we wanted to develop. We wanted to sort out the specific purpose that this application was to be used for. We needed to make sure that we did not try to add too many different types of functionality to this application or it would be confusing to the user what this application is truly used for unless the point of this application was to specifically have multiple functionality. This was not the case for our application. We came to the consensus that our application should have the main functionality of sorting a queue of online or in restaurant made orders so that a restaurant can get their orders done quicker to get food out to their customers in a reasonable time.

We started out in discussing the project abstractly taking in account the customer and restaurant interaction. We realized that though the interaction should be between

the customer and the restaurant there are many different aspects in between them that we would have to go to like orders and a menu that connect the customer and restaurant. This realization enabled us to take the project from an abstract view and formulate a concrete view of compartmentalizing our code into classes of order, menu, customer, and more. From here we started majorly developing the backend of the application while starting the GUI aspect of the application little by little.

We developed the backend using a map representation for the menu while using a linked list to represent the active orders sent in by customers to a restaurant. An Order Comparator class was made so that when adding and removing orders we could sort the queue to get the quickest orders to the front. The comparator also considered aging. If an order was in the queue while 5 orders have been accomplished we would increase the priority of that order that has not been accomplished yet. The classes provided standard procedures of adding and getting rid of menu items that really took the weight off the GUI aspect of the application since we did not have to worry about actually configuring the underlying structure correctly.

The GUI consisted of configuring the restaurant GUI that had two interfaces where first was where a restaurant could edit the menu we present to customers and second a restaurant could see the active orders coming in from customers. The second aspect of the overall GUI was the customer interface where they could see the menu a restaurant has posted and place orders. Developing this with the backend required a lot of communication between us as developers but the strong underlying back end helped us continuously add new features to make the application more robust and user friendly.

Overall our development process was organized and allowed us to constructively add feature after feature to make the program better and better for future users. It

enabled us to fully test the application when basic functionality was done and add great features that pushed the project to where we wanted it to be.

Conclusion

Overall our ninja fast kitchen system is a modern and robust kitchen display system applicable for a general size restaurant whether it be a small family business, or a large chain such as mcdonalds. Our display system accomplishes the functionality for businesses to create their own menus, customers to order from said menus, and most importantly, it allows restaurants to see the current active orders, as well as offers tips on how to complete those orders most efficiently. Our display system is also very scalable, if we had more time to develop, some features we would have finished are, adding more helpful hints on how to make orders most efficiently, such as how to pool items together to make multiple orders at once. Another feature we would have liked to develop is alerting a restaurant if an order has been sitting incomplete for a long amount of time. Another long term goal we had for this project was to convert the customer gui into a mobile app, which would have the functionality to scan a QR code to bring in a menu, this would make our ordering system more applicable to customers, and more effective in a practical restaurant setting. Another interesting addition to the project would be the ability for the restaurant to act as the master customer. They would be able to be constantly placing and editing orders for customers, thus taking user error from the customer out of the equation entirely. This would work great for family restaurants whose clientele may not be as well versed with computers as larger chains.