Milestone 6: Final Milestone Assignment

Submitted by

Group 11: Blazin_Seven
CMPT 370 (2017-2018)
University of Saskatchewan

Aparna Angu
Duke Rong
Emmanuel Oriade
Eric Li
Josh Kocur
Ridwan Raji
Yinsheng Dong

Submitted to

Dr. Chanchal Roy

Assistant Professor
Department of Computer Science
University of Saskatchewan

December, 7th, 2017, Saskatoon

Food Ordering System is a desktop application that can be used by customers and restaurants to encourage a more open and responsive relationship between the customer and the business. Customers can use this application to search for different versions of their favourite foods in their city. Furthermore, these searches can be organized by how far the restaurant is, how long the current wait time is, the prices of the food, and by how other customers rate the restaurant. Customers can also easily place orders and pay through the application which both the order and payment will be transferred to the restaurant. The production of this application was challenging due to having to not only incorporate many different components from many different members into one functional system but also because our members firstly needed to learn how to create these components. Throughout this product, we gained experience in user design, database creation and management, using external API libraries, and most importantly working effectively and efficiently in a large group. The following document showcases Blazin_Seven's work that went into creating FOS.

Table of Contents **Page No.**

1. Introduction (2 marks)

1.1 System description

FOS is an application where users can search for various food items or restaurants, order food from participating restaurants, and rate restaurants based on their food and service. The search results can be sorted by different metrics depending on what the user values. The users can sort food by "stars" (1 through 5), expected wait time, prices, and distance (between the user and the restaurant). The system will also allow customers to keep a list of their favourite restaurants and foods for easy ordering. Finally, the system will send notifications to the restaurant of an incoming order, to the customer when an order has been confirmed, and to both users as a confirmation of their account creation.

FOS utilizes a database where customer and restaurant information is stored. Information is extracted from the database when the customer executes a search and the system will sort the search results based on the user's chosen metric. Determining the distance between the restaurant and the user is achieved using Google's Distance Matrix API. Notification and confirmations are sent to users through email.

1.2 Business case

This system seeks to solve the problem of the uninformed, hungry, would-be diner. It will provide its users the ability to solve the question of "what is the best version of this food in the city?", "what is the cheapest version of my favorite food?", and even "what is the closest restaurant offering the food I am craving?" Offering solutions to these questions allows users to make informed decisions ensuring they are spending within budget, and are provided with the opportunities to eat the best version of what they desire most. Furthermore, this would be done all while knowing approximately how long a particular item will take to be prepared. From the perspective of a business, this system will allow for an increased consumer base without any extra effort resulting in increased profits without increased expenses.

1.3 User-level goals for the system

There are two primary users of the FOS application:

Customers:

- Use the product to browse different food options, including but not limited to different versions of the same food.
- Sort the same food item by price, expected wait time, rating, and distance.
- Place an order with the participating restaurant of their choice, determined by their food selection
- Create an account and keep a list of favourite foods for easy ordering.

Restaurants:

- Increase customer base by increasing the exposure of the restaurant's menu and

rating
- Have a centralized location for the restaurant's address, timings, and menu so that customers can easily find more information about the restaurant
- Receive more and streamline the orders of food through their application

1.4 User scenarios

Customers:

- Food/Restaurant Search
    - A customer, who wishes to see different pepperoni pizza options in the city, will use the search function to be presented with those options.
- Price Search
    - A customer, who is price conscious, wants to eat spaghetti for dinner. This user will use the search function and then sort by price to find something in their budget.
- Wait Time Search
    - A customer, who is impatient and hungry, wants to eat a cheeseburger as soon as possible. This user will use the search bar and sort the food options by wait time to find the applicable restaurant that will provide their desired food choice.
- Distance Search
    - A customer, who does not want to travel very far, wants to find the closest restaurant that offers smoothies. The user will use the search bar and sort the food by distance to show the restaurants closest to the user.
- Rating Search
    - A customer, who wishes to eat the best fried rice in the city, will use the search function to not only find their food but also sort through the food options by ratings.
- Save Favourite Foods
    - A customer who enjoyed a previous order wishes to reorder their previous meal. This user would use information stored on their account to view and then replicate a previous order.
- Order Food
    - A customer wants to save time by ordering food straight through the application. The system will notify the restaurant and the customer of the confirmation of the order and the customer can also pay through the application.

Restaurants:

- Increase Exposure:
    - A restaurant just beginning their business wants to increase awareness of their restaurant will use FOS show that customers gain more exposure to the new restaurant in their area.
- Increase Revenue:

    - A restaurant wants to increase their revenue by expanding their service to take orders online will use FOS so that customers can order food from the comfort of their homes.
- Streamline online orders:
    - A restaurant wants to improve their service will use FOS to see customers' ratings and comments and make changes to their service

1.5 Scope document

- A search function that will be used to search for different food items.
- A browse function that stems from the search function that displays to the user all available options for the food they searched. It will be displayed in a table.
- A function on the browse page that allows user to sort by the different metrics such as price, distance, rating, wait time
- A function that allows users to log into the system.
- An account page that keeps track of the user's information, favorite orders as well as a list of past orders by recent date.
- A function that places orders with the restaurant.
- A function that will send notifications to restaurants and customers
- A function that will allow customers to leave ratings and comments about their experiences

1.6 Project plan / Rough estimates

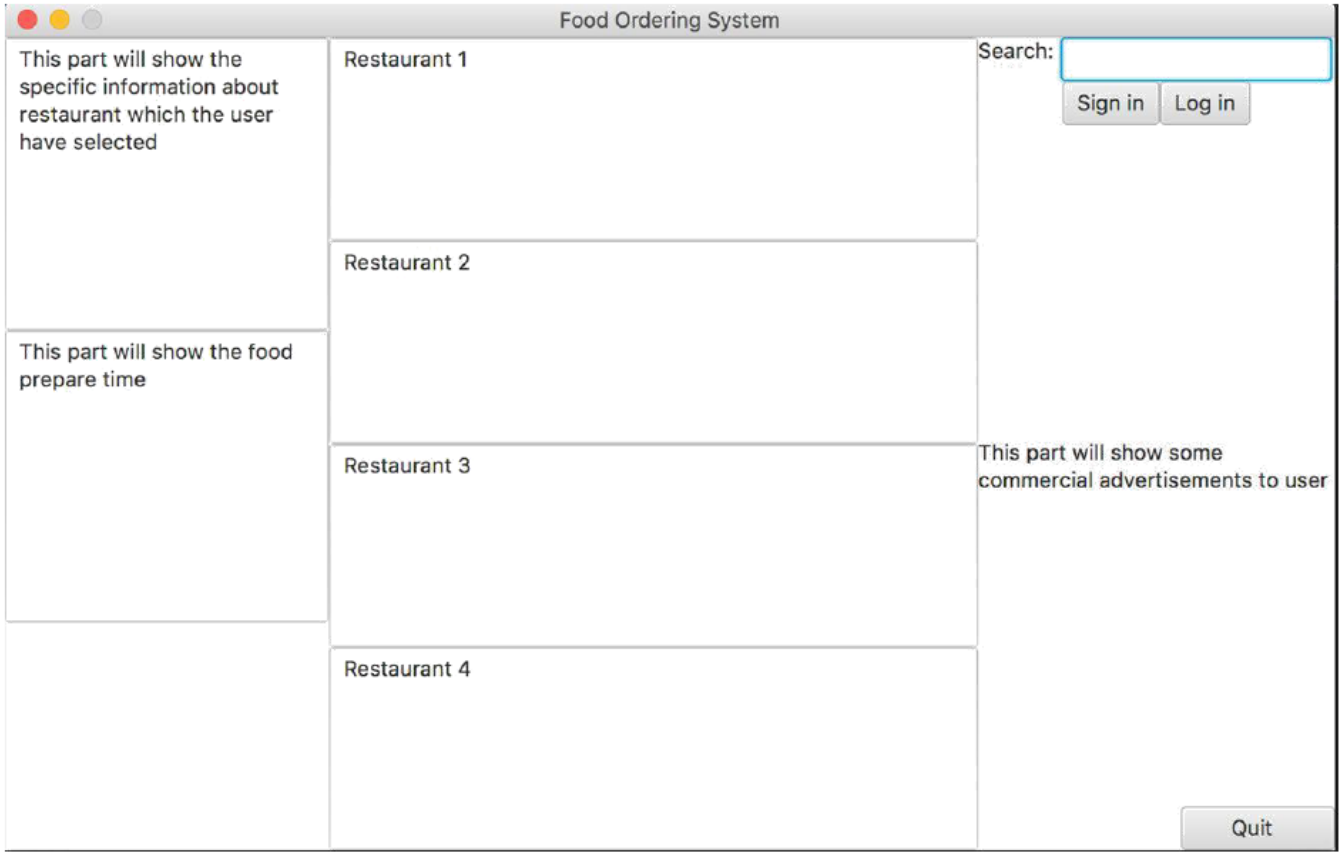
We endeavor to have completed a working, useable database connection (assigned to: Josh, Yinsheng and Ridwan. Expected time: 25 hours), the ability for a user to create an account (assigned to: Aparna and Emmanuel. Expected time: 10 hours), as well as a working search function (assigned to: Hao and Rong. Expected time: 15 hours).

1.7 User involvement plan

We plan on doing extensive testing ourselves. However, this testing will not cover every aspect of user experience, so we plan on using our friends to help with testing. We will let them try our project to make sure it's both working correctly and is user friendly. Problems that arise can be dealt with in an ongoing manner through a system of verbal feedback. We will involve them after we implement a new feature, as this process is ongoing. We expect no more than a couple hours of time during the entire process.

1.8 Low fidelity prototypes

Toy Prototype for Landing Page for the Application:



2. Requirements and Early Design (17 marks)

2.1 Summary Use Cases (2 +1 marks)

(a) List the names of all the use cases of the system and provide a two line description for each of them.

- Increase restaurant exposure:
  - Make customers easily find a restaurant if it contains what they are looking for.
- Review Customer Feedback:
  - Restaurants can improve their service by regarding the customers' ratings and comments on the food, atmosphere and service of the restaurant
- Restaurant Notification:
  - Inform restaurants of incoming orders through the application
- Distance Search:
  - Sort the results of the customers' searches by closest restaurant through

farthest restaurant

- Price Search:
  - Sort the results of the customers' searches by the restaurant with the cheapest prices
- Rating Search:
  - Sort the results of the customers' searches by the restaurant with the best customer ratings
- Wait Time Search:
  - Sort the results of the customers' searches by the restaurant with the lowest wait time for preparing the customers' food order
- Restaurant Search:
  - Search for a specific restaurant to know the locations, timings, and contact information
- Browse Restaurant Options
  - Customers can look through the application to see all the restaurants available in the customers' cities
- Adding Reviews
  - Customers can add reviews of their recent orders to comment on the restaurant's service and quality of food
- Place Order
  - Customers can directly place an order on the food that they have searched through the Food Ordering System application
- Pay for Order
  - Customers can pay for the food that they ordered through the application
- Account Creation and Login
  - Customers can create an account that will save the customers' information to prevent having to keep entering information during each search/order
- Save Favourite Foods
  - Customers can save favourite foods to their account which makes it easy for the customer to place an order on their favourite foods without having to search for the food first.

(b) Summary Use Cases

1) Increase Restaurant Exposure

Level: Summery
Actor: User - Restaurant
Goal: Increase exposure through advertisement
Activities: A ‘mom and pop’ style restaurant might wish to increase exposure and can do so by using this desktop application. This could be accomplished by offering new and unique food items that users might desire, or even by announcing their location through the sort by distance field.
Quality: This is a helpful feature for restaurants but is not the main use case. It should be working at a usable standard.
Version: October 5th, 2017

2) Reviewing Customer Feedback

Level: Summery
Actor: User - Restaurant
Goal: Improve quality of restaurant service
Activities: A participating restaurant can use user provided feedback to improve their menu options and restaurant experience. A restaurant can go to their account page and to a convenient customer comments section. From reading the comments, the restaurant can make the necessary changes. For example, if a particular item has been repeatedly deemed “not spicy enough”, changes could be made.
Quality: This is a helpful feature for restaurants but is not the main use case. It should be working at a usable standard.
Version: October 5th, 2017

3) FOS Restaurant Notification

Level: Summary
Actor: User - Restaurant
Goal: To inform participating restaurants of incoming orders
Activities: A customer ordering food from a participating restaurant will have their order and its information sent in real time the applicable restaurant. Expected wait times are provided to the user, and once the food is prepared, another notification is sent to the user.
Quality: Food Ordering is the main function of this program. This use case should be working to a very high standard.
Version: Oct 3rd , 2017

4) Distance Search

Distance Search
Level: Summary
Actor: User - Customer
Goal: To show the customer the closest restaurants along with their desired food item.
Activities: A customer is wishing to order a food item but does not want to travel far to pick up an item. The customer can browse for the nearest restaurants or they can enter their desired food item into the search bar and choose the metric option of distance. The search will return a list of restaurants with the desired food item sorted from closest to farthest.
Quality: This is a main use case in the system and should be working to a high standard.
Version: October 5th, 2017

5) Price Search

Level: Summary
Actor: User - Customer
Goal: To show the customer the cheapest prices of their desired food item.
Activities: A customer who is conscientious about money wants to search for the restaurants that are within their spending limit or they can search for the cheapest price of their desired food item. The customer will enter their desired food item into the search bar and choose the metric option of price. The search will return a list of restaurants with the desired food item sorted from cheapest to most expensive.
Quality: This is a main use case in the system and should be working to a high standard.
Version: October 5th, 2017

6) Rating Search

Level: Summary
Actor: User - Customer
Goal: To show the customer the best rated restaurants offering their desired food item.
Activities: A customer wants to find the best restaurants in the city or want to find the best version of their desired food item in the city. The customer will enter their desired food item into the search bar and choose the metric option of highest rated. The search will return a list of restaurants with the desired food item sorted from highest rated to lowest rated.
Quality: This is a main use case in the system and should be working to a high standard.
Version: Oct 5th, 2017

7) Wait/Prep Time Search

Wait Time Search
Level: Summary
Actor: User - Customer

Goal: To show the customer the restaurants with the lowest waiting time that has their desired food item.
Activities: A customer does not want to wait long to for their desired food item. The customer will enter their desired food item into the search bar and choose the metric of the lowest prep/wait time. The search will return a list of restaurants sorted from lowest wait time to highest wait time that have the customer's desired food item.
Quality: This is a main use case in the system and should be working to a high standard.
Version: Oct 5th, 2017

8) Adding Reviews

Adding Customer Reviews
Level: Summary
Actors: User - Customer
Goal: To allow user to add a review about a food item
Activities: A customer, who recently ordered and finished eating a food item wishes to add a review about the food quality. They can open the program, go to their recently ordered food items and easily add a review including a rating out of 5 stars and a comment accompanying their rating.
Quality: Ratings are one of the metrics that the system uses to help the customer to search for food. Thus, while the comment section is not a critical component, the five star rating is and thus work to a high standard.
Version: October 5th, 2017

9) Placing Orders

Level: Summary
Actor: User - Customer
Goal: To avoid queuing at the restaurant and make ordering food hassle-free.
Activities: A busy and hungry customer does not want to wait for their ordered food at a restaurant because of time constraints. The customer can search for a restaurant or their desired food item with their choice of a metric option and place an order. The order can be placed to prepare the food right away or to have it ready by an allotted time later on during the day.
Quality: This is a main use case in the system and should be working to a high standard.
Version: October 2, 2017

10) Payment Option

Level: Summary
Actor: User - Customer
Goal: To allow the customer to pay their bill through the program
Activities: When the customer finishes ordering their food, the customer may want to save time and pay through the program. They will find two options that will allow them to pay when the food is picked up or through their account. The customer will choose to

pay through the program and they can save/unsave their credit card information to their account. Now, the user can easily pay for their order.
Quality: Payment options will be made available. However, it is not the main use case of the program and will be working to a usable standard.
Version: October 1st, 2017

11) Save favourite foods

Level: Summary
Actor: User - Customer
Goal: To allow the customer to save their favourite foods
Activities: A customer has a regular food item that they frequently order. Instead of having to search for the item and then placing an order, the user can save their favourite foods to their account. Then in the future, they will simply have to open their account page, go to their saved items, and place an order.
Quality:
Version: October 7th, 2017

12) Browse Restaurant Options

Level: Summary
Actor: User - Customer
Goal: Allow a user to peruse different food options in their city.
Activities: A user may not wish to order food, but they may still want to know their options. They would use the search function with their desired metric to display different orderable options so that next time they do want to order food, they have a better understanding of what's available.
Quality:
Version: October 5th 2017

13) Search Restaurants Information

Level: Summary
Actor: User - Customer
Goal: Allow a user to search for a specific restaurant in the database
Activities: A user may want to look up information about a specific restaurant. Instead of searching for food items of the restaurant or finding the restaurant through the browser, the user can enter the restaurant's name in the search bar and quickly find the information that they are looking for. This information may include location, menu, open/close times, or ratings.
Quality: This use case will include the base function that all other searches will build from. Thus, it should be a very high quality.
Version: October 12th 2017

14) User Account Creation and Log In
Level: Summary
Actor: User – Customer & Restaurant
Goal: Allow customers to create accounts to save personal information and restaurants to join the program's database of restaurants.
Activities: A customer does not want to repeatedly enter their information each time an order is placed. Creating an account will allow them to save information such as name, location, credit card information, and favourite foods to make the process of ordering go faster. A restaurant wants to increase their exposure to new customers and will create an account to be included in the searches by the customers. In their account, information such as menu, location, open/close times will be added for searches.
Quality: Restaurant account creation is important for this program to exist. Thus, this should be working at a high quality.
Version: October 12th 2017

2.2 Fully-dressed Use Cases (1 marks)

Use Case 1: Search for Distance

Scope: Food Ordering System Application

Primary Actor: Customer

Stakeholders and Interests:
- Customer: Wants an easily readable display of restaurants or restaurants with their desired food item organized from closest to farthest to themselves.
- Restaurant: Wants to ensure that if a customer is within close distance to their restaurant, then their restaurant is appearing in the distance search results and their address is accurately listed.

Preconditions: The user should select the distance metric through the search bar for a food item or through the browser for restaurants. The customer's location is authenticated through their account information.

Success Guarantee and Post conditions: Search correctly returns a list of restaurants ordered from closest to farthest displaying the measure of distance in kilometers.

Main Success Scenarios:
1. Customer opens and logs into the application
2. Customer enters their desired food item into the search bar.
3. Customer chooses the metric of distance.
4. Program authenticates the customer's location.
5. Program will search the database for restaurants containing the customer's desired food item and return a list.
6. Program will organize the list of restaurants from closest to farthest by comparing the restaurant's location and the customer's location.

7. The program returns a list of restaurants containing the customer's desired food item ordered from closest to farthest to the customer in the UI.

Extensions:

2a. The customer is not looking for a specific food item and only wants to browse for restaurants closest to them:

1. Customer chooses the browsing option.
2. Program will search through the database for the restaurants that are closest to the customer
3. The program will sort the list from the closest to the farthest restaurants using the customer's location
4. The program returns the list of restaurants to the customer in the UI.

2b. Customer's desired food item is not in database

1. Return a statement saying item cannot be found

4a. The customer's location cannot be authenticated.

1. Customer's address location from account is used.

*a. At anytime the application crashes.

1. Customer closes and reopens the application.

Special Requirements:

- Any working operating system with the FOS desktop application downloaded

Technology and Data Variations List:

- Search on Mac OS using the .jar of FOS
- Search on Windows using the .exe of FOS

Frequency of Occurrence: Since it is the main function of the application, the distance search will be used often.

Open Issues:

- Will there be a limit or a max distance that the restaurants will be chosen from?
- Can the customer choose for the number of closest restaurants?
- Can the customer choose what the max distance should be?
- Can this metric be combined with other metrics?

Use Case 2: Search for Price

Scope: Food Ordering System Application

Primary Actor: Customer

Stakeholders and Interests:

- Customer: Wants an easily readable display of restaurants or restaurants with their desired food item organized from cheapest to most expensive

- Restaurant: Wants to ensure that if the restaurant contains the customer's desired food item, then they are included in the returned list to the customer along with their accurate information.

Preconditions: The user should select the price metric through the search bar for a food item or through the browser for restaurants.

Success Guarantee and Post conditions: Search correctly returns a list of restaurants ordered from cheapest to most expensive.

Main Success Scenarios:

1. Customer opens and logs into the application
2. Customer enters their desired food item into the search bar.
3. Customer chooses the metric of price
4. Program will search the database for restaurants containing the customer's desired food item and return a list.
5. Program will sort the list of restaurants from cheapest to most expensive
6. The program returns a list of restaurants containing the customer's desired food item ordered from cheapest to most expensive to the customer in the UI.

Extensions:

2a. The customer is not looking for a specific food item and only wants to search for cheap restaurants in the city.

1. Customer chooses the browsing option.
2. Program will search through the database for a list of restaurants
3. The program will sort the list from the cheapest to the most expensive restaurants
4. The program returns the list of restaurants to the customer in the UI.

2b. Customer's desired food item is not in database

1. Return a statement saying item cannot be found

*a. At anytime the application crashes.

1. Customer closes and reopens the application.

Special Requirements:

- Any working operating system with the FOS desktop application downloaded

Technology and Data Variations List:

- Search on Mac OS using the .jar of FOS
- Search on Windows using the .exe of FOS

Frequency of Occurrence: Since it is the main function of the application, the price search will be used often.

Open Issues:

- Can the customer choose the highest price to search under?
- Can the customer choose how many restaurants to get returned to them?

- Can the customer choose the range of prices to check for?
- Can this metric be combined with other metrics?

Use Case 3: Search for Ratings

Scope: Food Ordering System Application

Primary Actor: Customer

Stakeholders and Interests:
- Customer: Wants an easily readable display of restaurants or restaurants with their desired food item organized from the most highly rated to the lowest rated
- Restaurant: Wants to ensure that if the restaurant is positively rated, then they are included in the returned list to the customer along with their accurate information.

Preconditions: The user should select the rating metric through the search bar for a food item or through the browser for restaurants.

Success Guarantee and Post conditions: Search correctly returns a list of restaurants ordered from the highly rated to the lowest rated.

Main Success Scenarios:
1. Customer opens and logs into the application
2. Customer enters their desired food item into the search bar.
3. Customer chooses the metric of ratings
4. Program will search the database for restaurants containing the customer's desired food item and return a list.
5. Program will organize the list of restaurants from highest rating to the lowest ratings
6. The program returns a list of restaurants containing the customer's desired food item ordered from the highest rated to the lowest rated to the customer in the UI.

Extensions:

2a. The customer is not looking for a specific food item and only wants to search for the best restaurants in the city.
1. Customer chooses the browsing option.
2. Program will search through the database for a list of restaurants
3. The program will sort the list from the highest rated to the lowest restaurants
4. The program returns the list of restaurants to the customer in the UI.

2b. Customer's desired food item is not in database
5. Return a statement saying item cannot be found

*a. At anytime the application crashes.
1. Customer closes and reopens the application.

Special Requirements:

- Any working operating system with the FOS desktop application downloaded

Technology and Data Variations List:
- Search on Mac OS using the .jar of FOS
- Search on Windows using the .exe of FOS

Frequency of Occurrence: Since it is the main function of the application, the rating search will be used often.

Open Issues:
- Can the customer choose how many restaurants to get returned to them?
- Can the customer choose the range of ratings of restaurants to search from?
- Can the customer combine this metric with other metrics?

Use Case 4: Search for Waiting Time

Scope: Food Ordering System Application

Primary Actor: Customer

Stakeholders and Interests:
- Customer: Wants an easily readable display of restaurants or restaurants with their desired food item organized from lowest waiting time to highest waiting time
- Restaurant: Wants to ensure that if the restaurant contains the customer's desired food item and have a relatively low wait time, then they are included in the returned list to the customer along with their accurate information.

Preconditions: The user should select the wait time metric through the search bar for a food item or through the browser for restaurants.

Success Guarantee and Post conditions: Search correctly returns a list of restaurants ordered from lowest wait time to the highest wait time.

Main Success Scenarios:
1. Customer opens and logs into the application
2. Customer enters their desired food item into the search bar.
3. Customer chooses the optional metric of wait time
4. Program will search the database for restaurants containing the customer's desired food item and return a list.
5. Program will sort the list of restaurants from lowest wait time to highest wait time
6. The program returns a list of restaurants containing the customer's desired food item ordered from lowest wait time to highest wait time to the customer in the UI.

Extensions:

2a. The customer is not looking for a specific food item and only wants to search the restaurants with the lowest wait time in the city.

1. Customer chooses the browsing option.
2. Program will search through the database for a list of restaurants
3. The program will sort the list of restaurants from the lowest wait time to the highest wait time
4. The program returns the list of restaurants to the customer in the UI.

2b. Customer's desired food item is not in database

5. Return a statement saying item cannot be found

*a. At anytime the application crashes.

1. Customer closes and reopens the application.

Special Requirements:

- Any working operating system with the FOS desktop application downloaded

Technology and Data Variations List:

- Search on Mac OS using the .jar of FOS
- Search on Windows using the .exe of FOS

Frequency of Occurrence: Since it is the main function of the application, the wait time search will be used often.

Open Issues:

- Can the customer choose how many restaurants they want to be returned to them?
- Can this metric be combined with another metric?
- Will the metric use average prep time or will they have the restaurant's wait time have real time updates?

Use Case 5: Search Restaurant Information
Scope: Food Ordering System Application
Primary Actor: Customer
Stakeholders and Interests:

- Customer: Wants to generate information about the restaurant their interested in
- Restaurant: Wants their information, such as their location and menu, to be accurate and to be easily accessible to the customer.

Preconditions: Customers input the restaurant name in the search bar
Success Guarantee and Post conditions: Related restaurant name and information will appear in the UI if the restaurant is in the database.
Main Success Scenarios:

1. Customers input the name or some related words into the search bar and press "go"
2. Restaurants which fit the situation will show up at the center of the main interface.
3. Customers press the name of the restaurant to generate more information.

Extensions:

2a. The restaurant is not exist in our database

1. The customer will be notified that there is no restaurant matching their description in the database.

2b. The customers use different upper and lower cases which may return no or unwanted results

Special Requirements:

- Any working operating system with the FOS desktop application downloaded

Technology and Data Variations List:

- Search on Mac OS using the .jar of FOS
- Search on Windows using the .exe of FOS
-

Frequency of Occurrence: Many times, a customer just wants to look at the restaurants information. This use case will be used frequently because it is inconvenient to search for a food item of the restaurant or navigate through browsing options to find a restaurant's information.

Use Case 6: Placing an Order

Scope: Food Ordering System Application

Primary Actor: Customer

Stakeholders and Interests:

- Customer: Wants to place an order of their desired food item of a restaurant and wants confirmation that the order is confirmed. Paying customers want to be ensured that the correct amount of money has been received by the restaurant.
- Restaurant: Wants to be notified of the incoming order in real time. Details such as which food items, the time when they should be prepared by, and the customer's information (name, order number, payment information) should be also be received and be accurate.
- Wants to ensure that money has been received by customers who paid through the application.

Preconditions: Customer must choose one or more food items to order

Success Guarantee and Post conditions: The correct order is placed. Money transacted through the application is validated. Both the customer and the restaurant receive their respective confirmation notifications. Restaurant begins to prepare the customer's desired food item.

Main Success Scenarios:

1. Customer opens and logs into the application

2. Customer enters their desired food item into the search bar and their choice of a metric option.
3. Program will search the database for restaurants containing the customer's desired food item and return a sorted list to the customer.
4. The customer can choose their food item and click a button to place an order on it
5. The customer will choose the option of having the food prepared right away
6. The restaurant will be notified of the incoming food order in real time
7. The customer will choose the option of paying through the application
8. The payment will be validated.
9. The customer will get a confirmation notification of their order being processed.

Extensions:

4a. Customer may want to pick up their order at a later time in the day instead of being picked up right away.

1. The customer will choose the option of picking up their order at a later time.
2. The customer will select the exact time at which they would like their order to be ready by.

5a. Customer may decide to pay at the restaurant

1. The customer will choose the option of paying at the restaurant
2. When the order is picked up, the restaurant will receive the payment in person.

6a. The order does not go through and the restaurant does not receive the notification

1. If the customer does not receive a confirmation notification, this indicates that the order did not go through.
2. The customer can try to place the order again
3. If problem persists, the customer can contact the system administrators.

8a. The payment from the customer does not successfully get transacted to the restaurant

1. If the customer does not receive a confirmation notification, this indicates that the order did not go through.
2. The customer can try to place the order again
3. If problem persists, the customer can contact the system administrators.

9a. The order goes through, both the customer and restaurant get confirmation notifications, but the restaurant does not prepare the food in time.

1. Customer should relay this information in the review section
2. System administrators can implement a strike system to discourage restaurants from preparing orders late.

*a. At anytime the application crashes.

1. Customer closes and reopens the application

Special Requirements:

- Any working operating system with the FOS desktop application downloaded

Technology and Data Variations List:

- Search on Mac OS using the .jar of FOS

- Search on Windows using the .exe of FOS

Frequency of Occurrence: Placing an order is one of the main functions of the application and will be used frequently.

Open Issues:
- How many strikes can a restaurant have?
- Will there be a compensation for the customer?

Use Case 7: Account Creation and Logging into Account
Scope: Food Ordering System Application
Primary Actor: Customer and Restaurant
Stakeholders and Interests:
- Customer: Wants program to save their information including name, address, credit card information, and favourite foods.
- Restaurant: Wants to include their restaurant in the searches to increase exposure to their restaurant. Wants address, menu, opening/closing hours, and pictures to be added to their account information.

Preconditions: Users without an account should use the "sign up" feature and users with the account should use the "log in" feature.
Success Guarantee and Post conditions: Customers and Restaurant users should be able to successfully log into their account with correct password. Their account information is saved will continue to appear each time users log into their accounts.
Main Success Scenarios:
1. Customer without an account will create one by pressing the "sign up" button on the landing page.
2. Customer will register by entering their username, password, and the, "Sign Up as a Customer" feature will be chosen.
3. The customer can complete their registration by clicking the button that says "Sign Up Right Now!"
4. Customer will receive a notification that the registration was a success.
5. Customer can now enter their personal information such as name, address, credit card information and save their favourite foods.
6. The customer's information will be saved to the application's database.

Extensions:
1a. The customer has already created an account.
   1. The customer can log in to their created account
   2. Customer can add/change their personal information
   3. Customer can log out.

1b. A restaurant without an account will create a new account.
   1. The restaurant should press the "sign up" button on the landing page.
   2. The restaurant should register with username, password, restaurant ID and the "Sign Up as a Restaurant" feature will be chosen

3. The restaurant will receive a notification that the registration was a success
4. The restaurant can proceed to add information to their account such as address, open/close times, menu, pictures etc.
5. The restaurant's information will be saved to the application's database.

1c. The restaurant has already created an account.

1. The restaurant can log in to their created account
2. Restaurant can add/change their personal information
3. Restaurant can log out.

2a. The customer username has already been used

1. Customer will get a notification indicating that the entered username has already been used.
2. The customer will enter a different username and can continue to do so until they create a unique username.

2b. The restaurant username and ID has already been used

1. Restaurant will get a notification indicating that the entered username or ID has already been used.
2. The restaurant will enter a different username and can continue to do so until they create a unique username. Or the restaurant must recheck their restaurant ID.

Special Requirements:

- Any working operating system with the FOS desktop application downloaded

Technology and Data Variations List:

- Search on Mac OS using the .jar of FOS
- Search on Windows using the .exe of FOS

Frequency of Occurrence: Customers will prefer to log-in into their account to skip entering their information such as their address and phone number and to save time. Thus, the log-in feature will be used often.

Open Issues:

- How will the program deal with customers registering as restaurants and vice versa?
- What if the customers/restaurants forget their username and/or password?

Use Case 8: Giving Reviews

Scope: Food Ordering System Application

Primary Actor: Restaurant

Stakeholders and Interests:

- Customer: Wants to give a review that includes a rating out of five stars and a comment that relays information about the restaurant such as the restaurant's food, staff, atmosphere, cleanliness, actual wait time, etc.

- Restaurant: Will want to their customers to give reviews of their service to improve their rating and get more exposure through the Rating Search and/or to see how they can further improve their service to their customers.

Preconditions: Customer must have ordered, picked up, and eaten their food from a restaurant available in this application

Success Guarantee and Post conditions: Order is placed and picked up by customer. Customer eats their food item and will give a review of the food through their account page under recent orders. The review information from the customer will be integrated with the current information of the restaurant and the ratings will be recalculated.

Main Success Scenarios:

1. Customer finds their desired food item and places the order.
2. Customer picks up the food item, experiences the restaurant and their customer service.
3. Customer eats the food item and wants to give a review of what they ordered and of their experience of the restaurant.
4. Customer goes to their account page, to recent orders, and chooses the review option.
5. Customers give a rating out of 5 stars and gives a comment explaining their choice.
6. Rating and comment information will be taken and the Restaurant's current rating will be recalculated

Extensions:

2a. Order may be placed but food is not picked up

1. Customers will be given a strike system that will discourage them from not picking up orders and wasting food. Too many strikes could ban the customer from placing orders.

5a. Customers may give a review that is offensive, derogatory, or unrelated to the restaurant's service.

1. Customer's reviews will quickly be screened checking for specific words that could prevent such comments.
2. If a customer's review contains these words, the customer's review will not be posted and may get a strike. Too many strikes could ban the customer from giving reviews.

Special Requirements:

- Any working operating system with the FOS desktop application downloaded

Technology and Data Variations List:

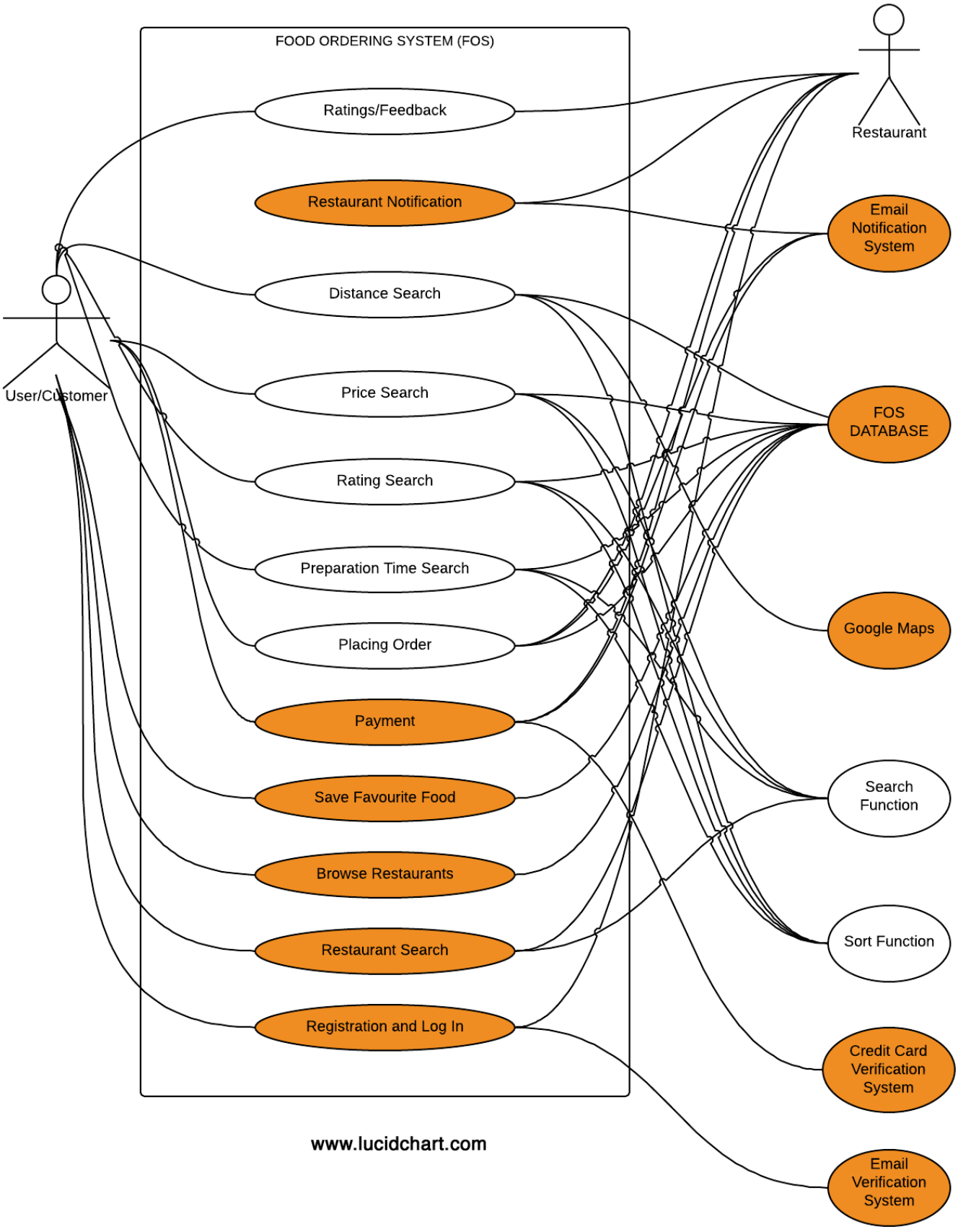
- Search on Mac OS using the .jar of FOS
- Search on Windows using the .exe of FOS

Frequency of Occurrence: The Rating Search is completely dependent on customers giving their review of restaurants. Without the customers giving reviews, this search would not work well. Thus this use case should occur often, hopefully as often as the number of orders placed.

Open Issues:

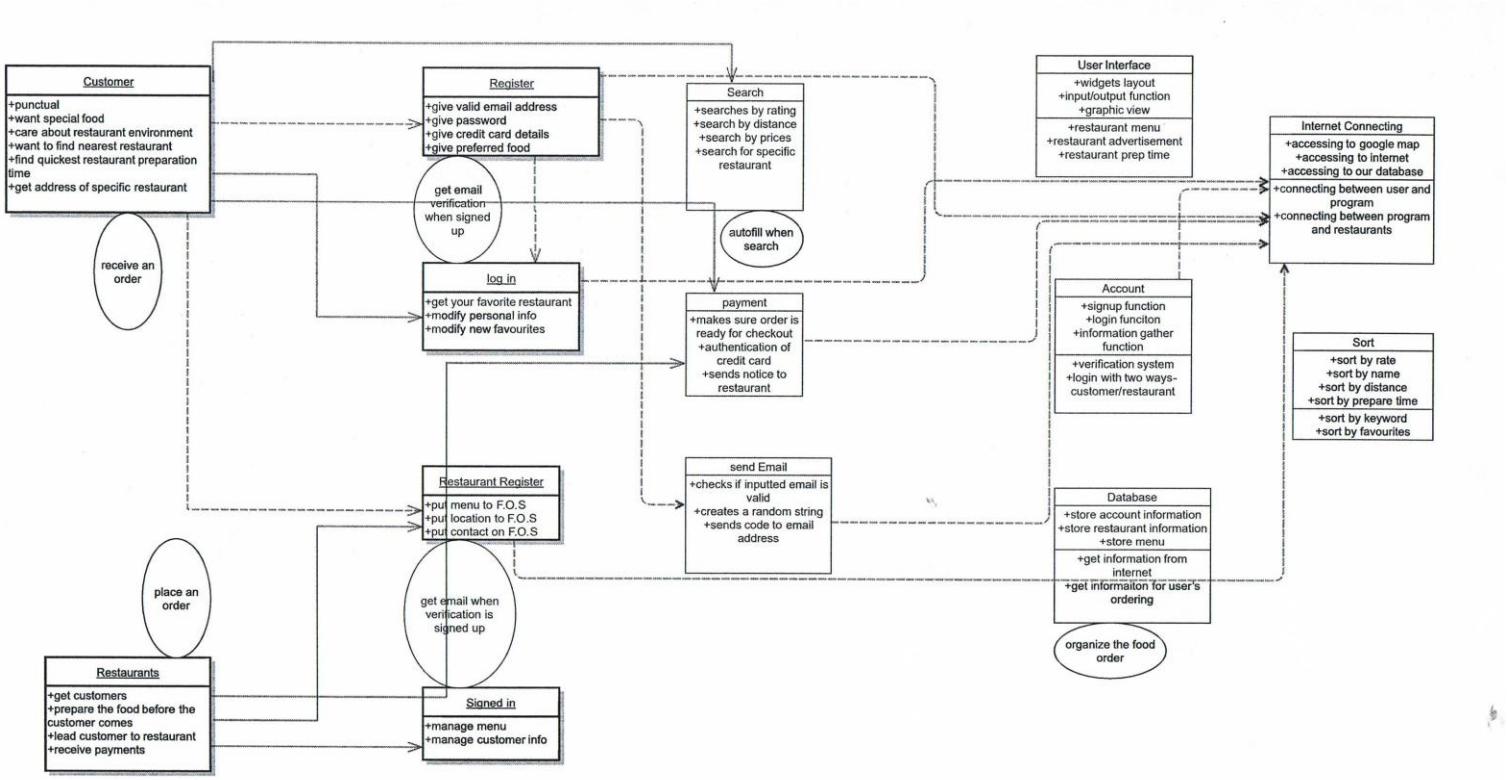
- Is there a way to help encourage customers to place reviews (coupons, point system, some rewards)
- Will there be a way to show if the restaurant that has been ordered from is on the customer's preference list with the review.
- Could unpicked orders donated/distributed to homeless shelters?

2.3 Use Case Diagram (1 mark)



** orange are newly added componenets

2.4 Domain Model (2 marks)



** For a clearer diagram, please see attached Domain Model in file

This is a diagram that shows the abstraction of the entire system, showing the various classes and their functionalities and how they link with one another.

This diagram of the entire Food Ordering System is an abstraction of the class diagram while showing the main functionalities of the system.

The User diagram is the class that consists of both the customer and the restaurant. It states the uses and purposes of it

The restaurant diagram is the class that prepare the food, takes payments.

The Signed in diagram is for the restaurant to manage and edit their menu and to manage their customer information.

The short chained/dash lines indicate that the class the arrow leads from is dependent on the class the arrow leads to. If something is changed in the class it leads to, the class it leads from is affected but not vice versa.

The classes that have the straight arrow lines show that they are related.

2.5 Glossary (2 marks)

AddCustomers: This is a class in the database package that deals with adding a new customer to the database of the customers.

AddDishes: This is a class in the database package that deals with adding a new dish to the database of the dishes.

addEmail: This is a function that takes in the email as a string. It takes in email as the parameter and it returns nothing.

AddLocation: This is a class in the database package used to insert location information for a user.

addLocations: This is a function in the AddLocation class in the database package. This function is used to add a new location. The parameters are user_id, house_num, street, city, provice, post_code and they are for the location information.

AddMenu: This is a class in the database package used to add menu of a restaurant.

AddMenu: This is a function in the AddMenu class in the database package. This is a function used to add menu information using restaurant_id and menuName which are the parameters.

AddOrderLines: This is a class in the database package that deals with tracking the number of orders placed for meals.

AddRestaurants: This is a class in the database package that adds restaurants to the system database.

AddUser: This class in the database package that is used to add a user, this should be before add customer and restaurant.

addUser: This is a function in the AddUser class in the database package. This is a function that is used for inserting a user to the database using username, userPassword, status as parameters.

CCNumberVerification: This is a class in the payment package that verifies the validity of the credit card number entered during payment. It has 2 global variables called CCNumber which is of type string and STD_CCN_DIGITS which is of type int.

Connect: This is a function in the GoConnection class in the database package. This is a function that is of type Connection, this function connects to the database.

Credit: This is a class in the payment package that takes in the card details and checks if the date is a valid date necessary for use.

Customer: This is a class in the Entities package that has the customer name, phone number, email, and other identification details.

CVVCodeVerification: This is a class to verify the validity of CVV Code of the Credit Card.

DateVerification: This is a class in the payment package that is used to verify the validity of Expiry Date of the Credit Card.

displayAllMenus: This is a function in the DisplayMenus class in the database package. This is a function displays menus in the database on the console. The function queries the sql database for the name of all food in the database. The function returns nothing.

DisplayDishes: This is a class in the database package that is used for the view of the different dishes available.

displayDishes: This is a function in the DisplayDishes class in the database package. This is a function that prints all the dishes unto the console. This function returns nothing.

displayItem: This is a function in the DisplayMenus class in the database package. This function prints out all items for a given menu. The function has menuID as the only parameter.

DisplayMenus: This is a class in the database package that shows all the menus on the console.

DisplayRestaurant: This is a helper class in the database package that helps to check every restaurant in the sql database.

displayRestaurantFromID: This is a function in the DisplayRestaurant class in the database package. This is a class function that displays the restaurant derived from the ID using restID which is the only parameter.

execute: This is a function that returns a list of Ids. It takes in the input, connection, getInt as its parameters.

executeName: This is a function that is called when the output is a list of string. It has input, connection, getString as the parameters.

ExecuteNoInput: This is class in the Search_Sort package is called when the search object is null.

ExecuteWithInput: This is a class in Search_Sort package that works when the search object is not empty.

findRestaurantSameCity: This is a function in the NearbyRestaurant class in the database package. This is a function dimply used to get the restaurant nearby (in the same city) a customer. It gets the restaurants found into a 2-d array and using customer_id as its only parameter.

Food: This is a class in the Entities package that has meal names and prices and preparation time.

FOS: The Food Ordering System is a system that provides an easy way to access different restaurants and their different menu and helps to place an order for the particular meal.

getCustomerAddress: This is a function in the NearbyRestaurant class in the database package. This is a function that is used to find the customer address, so we can use google function to calculate the distance between them. It uses customer_id as its only parameter and it returns nothing.

getDishesID: This is a function in the Food class in the Entities package. Returns the dish Id.

GetDistance: This class in the Google package is using Google API to get distance between 2 positions.

getDistanceString: This is a function in the GetDistance class in the google package. This is a function that returns the string of distance between origin and destination. It takes origin and destination as the parameters.

getEmail: this is a function that returns the email inputted.

getEmail: This is a function in the Customer class in the Entities package. This function returns a string of the email of the user.

getFirstName: This is a function in the Customer class in the Entities package. This function returns the first name of the user.

getFloatDistance: This is a function in the GetDistance class in the google package. This is a function that returns the float of a distance between origin and destination. It takes origin and destination as the parameters.

getID: This is a function in the Customer class in the Entities package. This function returns the customer id.

getLastName: This is a function in the Customer class in the Entities package. This function returns the last name of the user.

getName: This is a function in the Food class in the Entities package. This function returns the name of a particular food item.

getName: This is a function in the ExecuteWithInput class that generates the complete name of the food the customer searched for. This function has Query, connection, search, **getString** as its parameters and its returns the string generated from the database.

getPhoneNumber: This is a function in the Customer class in the Entities package. This function returns a string of the phone number of the user.

getPrefFood: This is a function in the Customer class in the Entities package . This function returns the food preparation.

getPrepTime: This is a function in the Food class in the Entities package. This function returns the preparation time for a given food item. It returns nothing.

getPrice: This is a function in the Food class in the Entities package. This function returns the price.

getUser_id: This is a function in the Customer class in the Entities package. This function returns the user id.

getRestaurantId: This is a function in the AddMenu class in the database package. This is a helper function to check the restaurant_id by using license_id. It returns an integer of the restaurant id and has license_id as its only parameter.

getRestaurantName: This is a function in the DisplayRestaurant class in the database package. This is a function that gets the restaurants name using the restID which is the only parameter.

GoConnection: This is a class in the database package that is used to make a connection to the sql db.

insertDatabase: This is a function in the Customer class in the Entities package. This function has the ability to insert customer, calls method from another package.

isCCNumberLengthValid: This is a function in the CCNumberVerification class in the payment package. This function checks the validity of the entered credit number using the length of the numbers entered.

isCCNumberValid: This is a function in the CCNumberVerification class in the payment package. This function checks the validity of the entered Credit Card number by checking if the numbers are actual integers.

isValid: This is a Boolean function that takes in the string email as a parameter and this function makes sure that the email is of a valid type.

Login: This is a class in the database package, and it main purpose is to make a log in function, help to check if the username and password that the user input are correct or not.

login: This is a function in the Login class in the database package. This is function that checks the username and password, it has the username and password as the parameters. This function returns nothing. --?

messageCode: This is a function that returns a randomly generated message/code of type string of 7 characters.

NearbyRestaurant: This is class in the database package used for finding the nearest restaurants.

queryForRestaurantName: This is a function in the DisplayRestaurant class in the database package. This is a helper function that select the query for the restaurant name.

restaurantNearBy: This is a function in the NearbyRestaurant class in the database package. This is a function to return the restaurant name and distance for a customer by a 2-d array. It has customer_id as its only parameter and returns an array list of the restaurants nearby.

SearchDishes: This is a class in database package used to query the dishes table for a dish with the name matching the string passed.

SearchMetric: This is a search filter that filters through the system database depending on the customer's interest. This search metric could either filter via distance metric, ratings metric, price metric, and waiting time metric.

SearchRestaurant: This is a function in the SearchRestaurants class in the database package. This is a function to search a restaurant by a name that given by users using restaurantName as a parameter. The search cannot be empty or null. This function returns nothing.

SearchRestaurants: This is a class in the database package used to search a restaurant by name.

Send: This is a function that takes in the returned string from the messageCode function and the email. It sends the messageCode and sends it to the email. This functions returns nothing.

setCustomerInfo: This is a function in the AddCustomers class in the database package. This function is used to add a customer's information to the database. It contains firstName, lastName, phone number, email and prefFood as its parameters.

setDishInfo: This is a function in the AddDishes class in the database package. This function is used to add a new dish information. The parameters are menus_id, dish_name, dishe_price, pptime and they are the dish information.

setFirstName: This is a function in the Customer class in the Entities package. This function has first name as the only parameter and sets the first name to the user.

setLastName: This is a function in the Customer class in the Entities package. This function has the last name as the only parameter and sets the last name to the user.

setName: This is a function in the Food class in the Entities package. This function sets a new name to new food item. It returns nothing.

setNewPrefFood: This is a function in the Customer class in the Entities package. This function has a food as the only parameter and sets the preparation time to the new food.

setOrderLineInfo: This is a function in the AddOrderLines class in the database package. This is a function sets the order line information. It returns nothing. This function has order_id, dishes_id, quantity, pricePerUnit, priceTotal, discount_total as the parameters.

setPhoneNumber: This is a function in the Customer class in the Entities package. This function has a new number as the only parameter and sets the new phone number to the user.

setPrepTime: This is a function in the Food class in the Entities package. This function has newPrepTime as the only parameter. It sets the preparation time and returns nothing.

setPrice: This is a function in the Food class in the Entities package. This function sets a new price to the food item. It returns nothing and throws an error if the price is set at negative.

setRestaurantInfo: This is a function in the AddRestaurants class in the database package. This is a function that is used to insert the parameters for setting a restaurant information using restName, licenseId, openTime, closeTime, phone_num, emailAddress as the parameters.

SignUp: This is a class that deals with the verification of the email provided by the customer. It sends a random code to the customer's provided email address.

Sort: This is a class in Search_Sort package that helps to filter and sort searches and results either by rate, distance, price, waiting time.

SortByDistance: This function sorts result by distance from the user using the search as parameter.

SortByPrice: This function sorts results by the price either ascending or descending order depending on the preference of the customer. This function has the array list of input result as the parameter.

SortByRate: This function sorts result by rate, using search as a parameter.

SortByWaitingTime: This function sorts results by the waiting time for each dish from fastest time to latest time depending on the preference of the customer. This function has search as the parameter.

System Diagram: this is a model used to visually express the dynamic forces acting upon the components of a process and the interactions between those forces.

System Sequence Diagram (SSD): This is a sequence diagram that shows, for a particular scenario of a use case, the events that external actors generate, their order, and possible inter-system events.

2.6 Supplementary Specification (2 marks)

1. Security
- Our system is set up in a way that browsing and searching through restaurants and their menus do not require user authentication but placing orders does require user authentication. This means that our system users do not need an account to look up restaurants but do need an account on our system to make an order and save the food preferences.

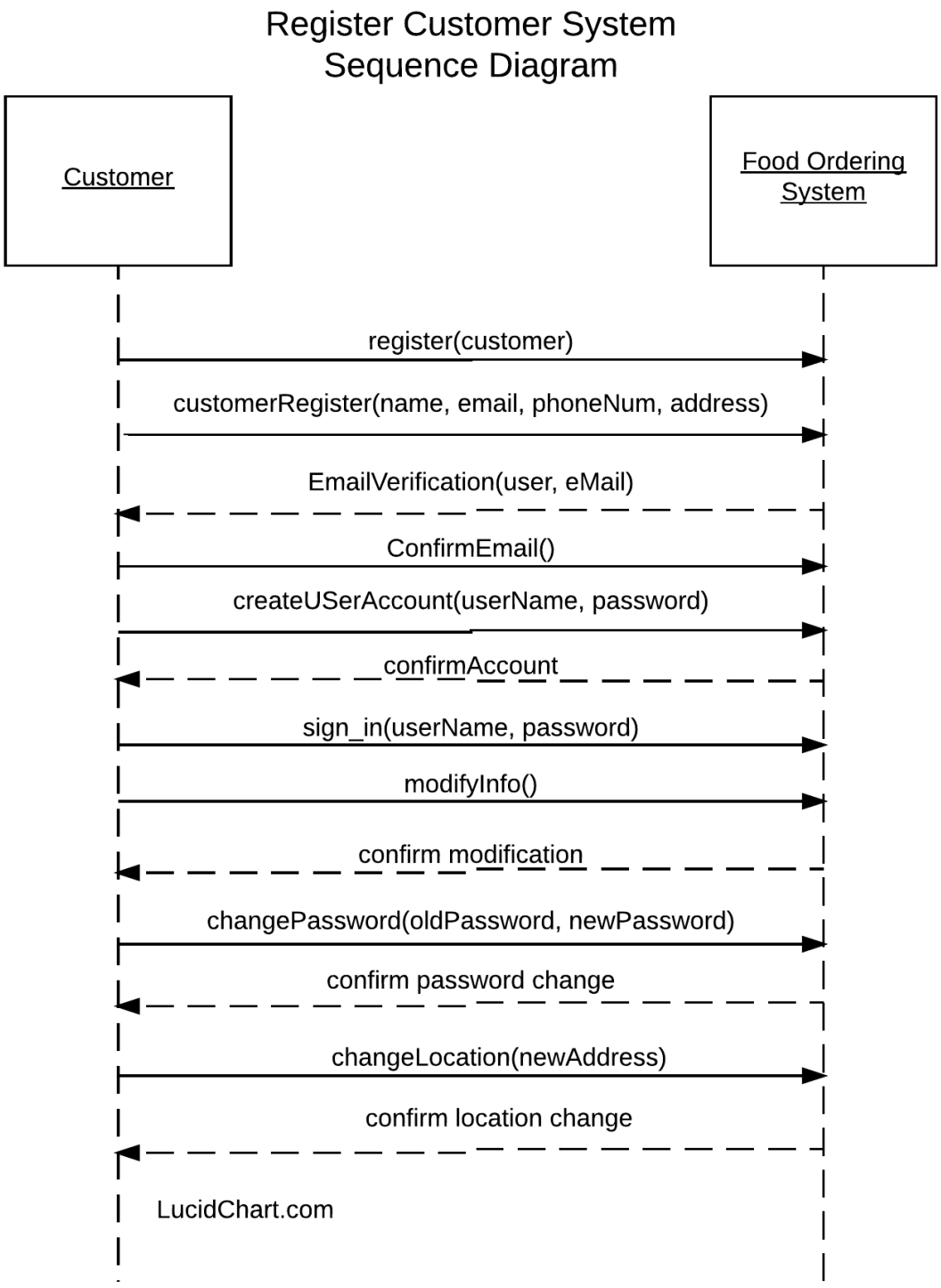
2. Human Factor
- Restaurant/Customer will be using the FOS (Food Ordering System) software on different displays. Therefore our system text and windows design will be proportional and legible for all screens on all different display sizes.
- We will also be avoiding colors associated with color blindness and all other common form of eye defects.
- Speed, easy usability and error-free processing are also important for our system.
- Our system will have an appealing design including pictures and graphics. These visuals are important in facilitating our system users to make their food ordering decisions.

3. Implementation Constrains
- We are using Java, JavaFx and SQL technologies to develop our system as these are the most widely known languages known by our group members. We are predicting it will help us improve our systems in the long term with porting, supportability, and also to ease further development in future.

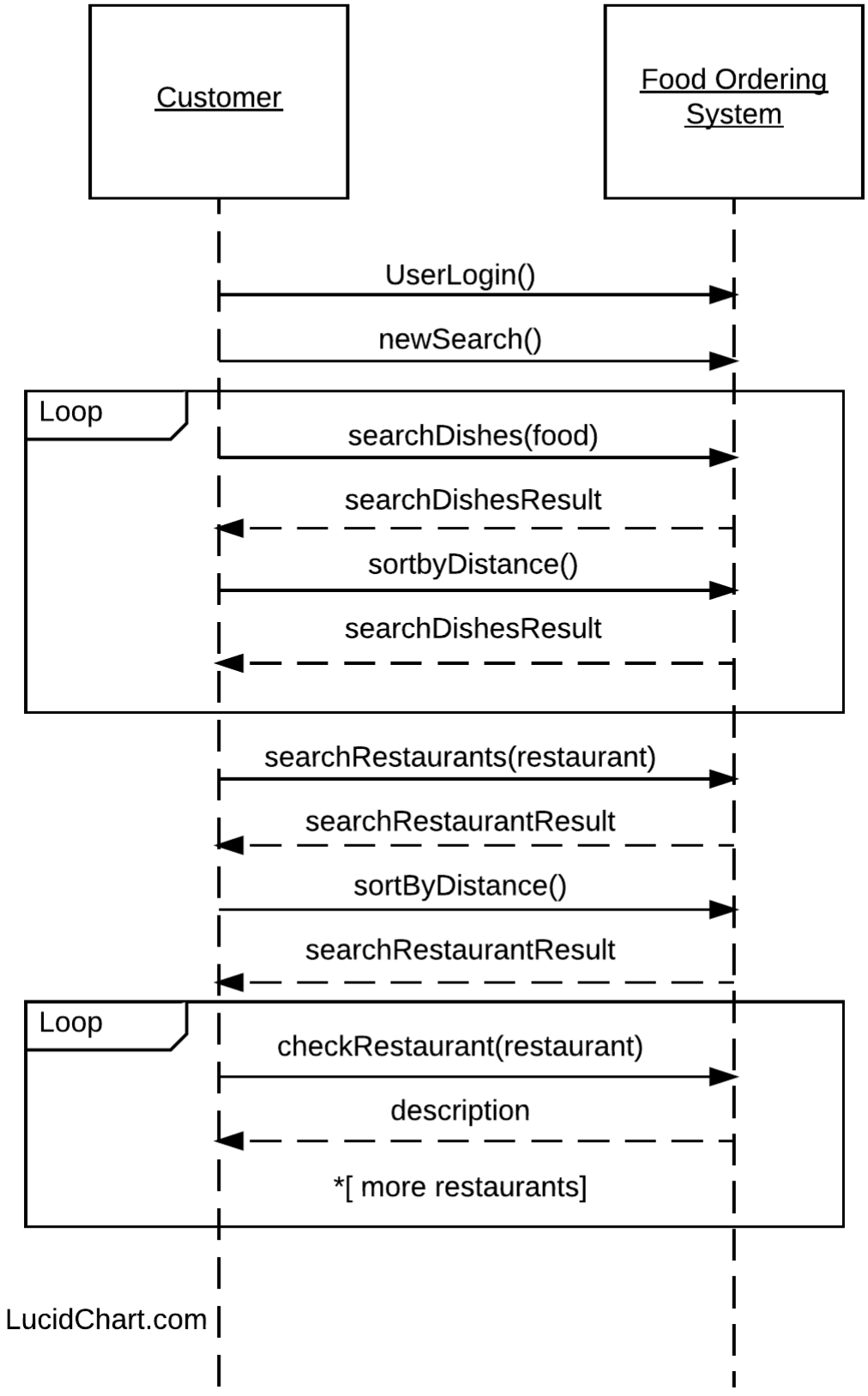
4. Free Open Source Component
- We will also be using some open source java technology components on this project. Though we have not completely decided all the open source components we will be using, one definite component that we will be using is the integration of Google Maps into our system to generate information about the distance between our system user and the restaurants they are browsing.

5. Software Interface
- Since we will be incorporating software like Google Maps and SQL Database into our system, we will be implementing our interface in such a way that all the stated above incorporation will be simple and hassle free.

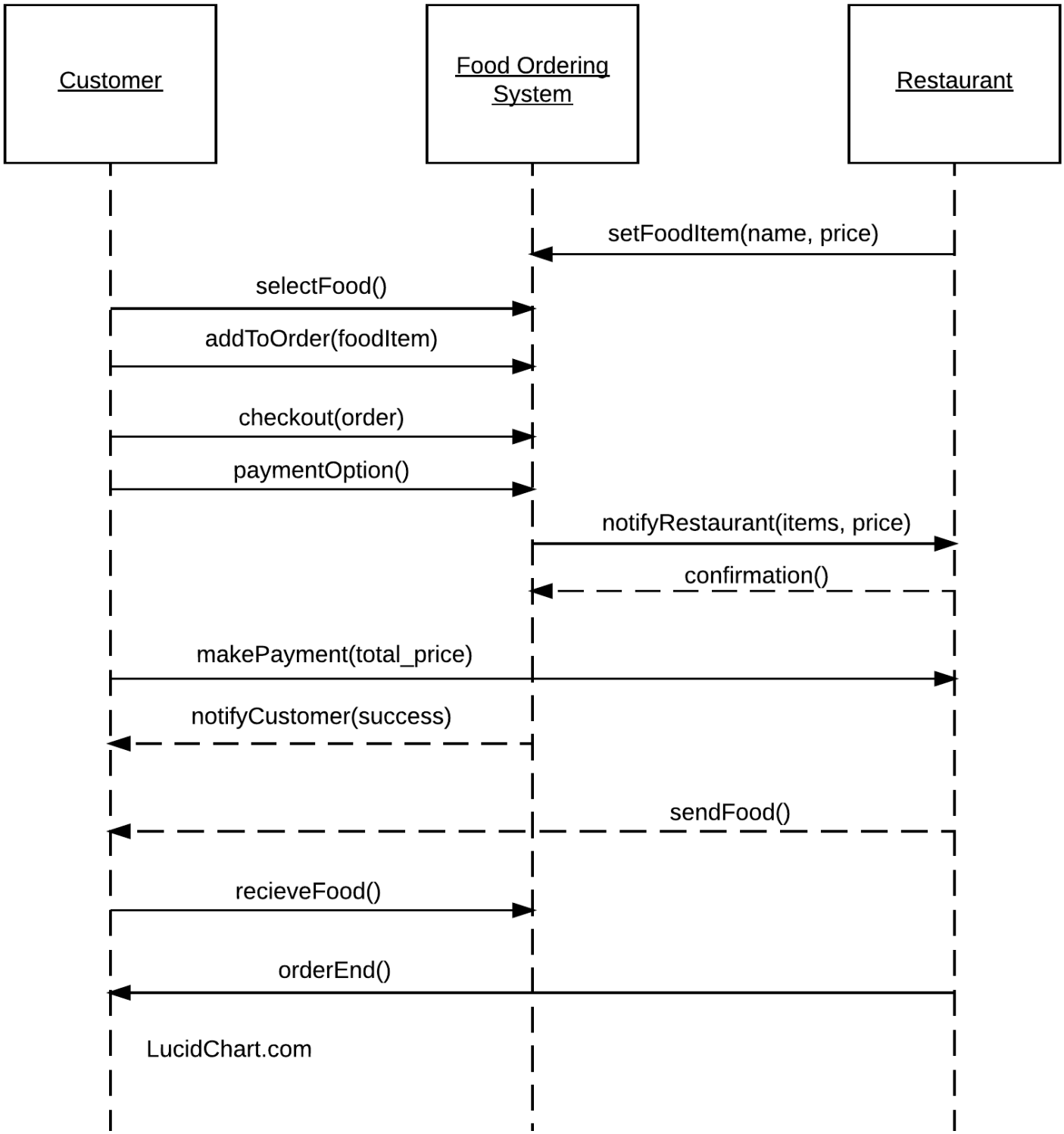
2.7 System Sequence Diagrams (3 marks)



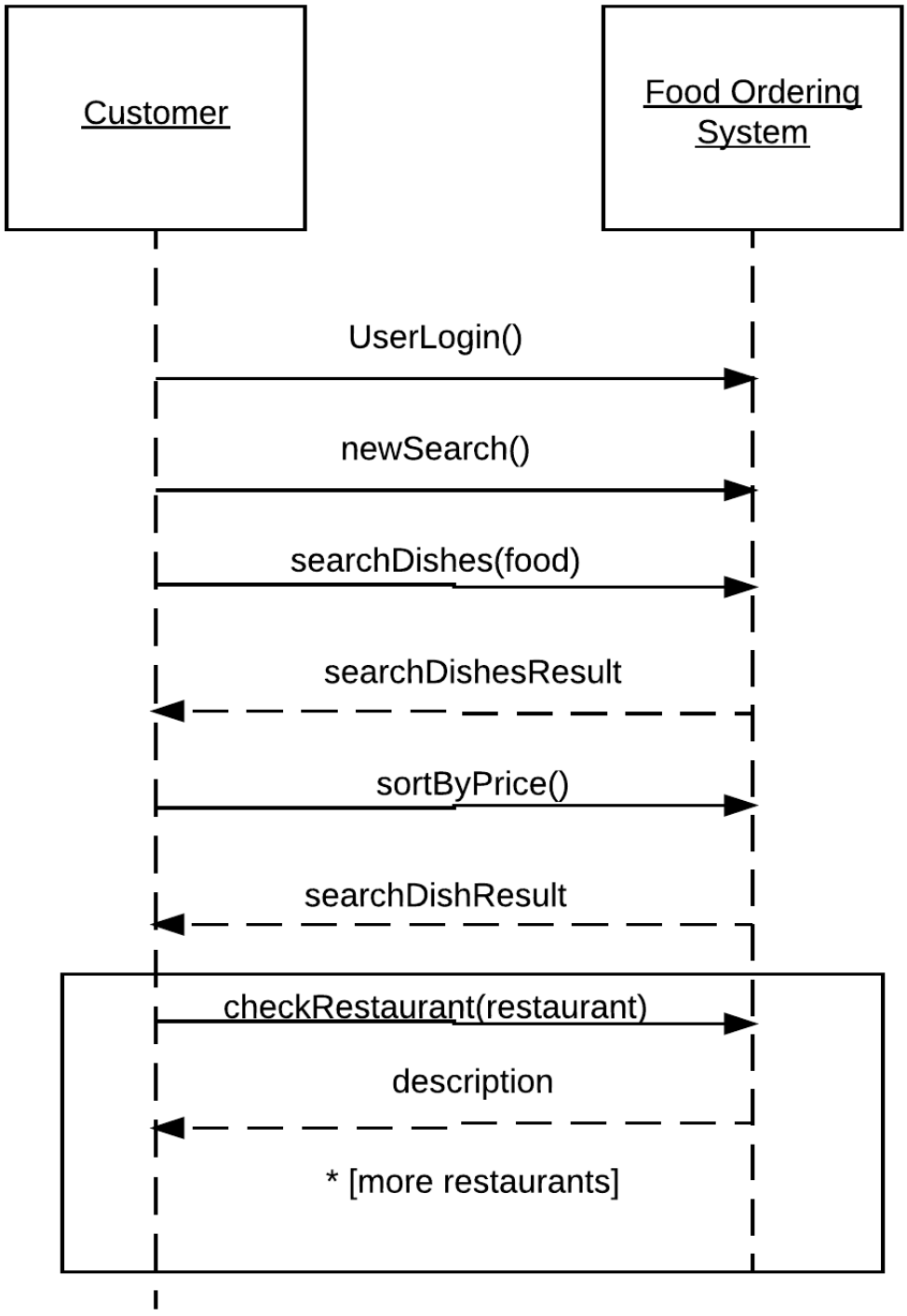
Distance Sort System Sequence Diagram



Placing Orders System Sequence Diagram

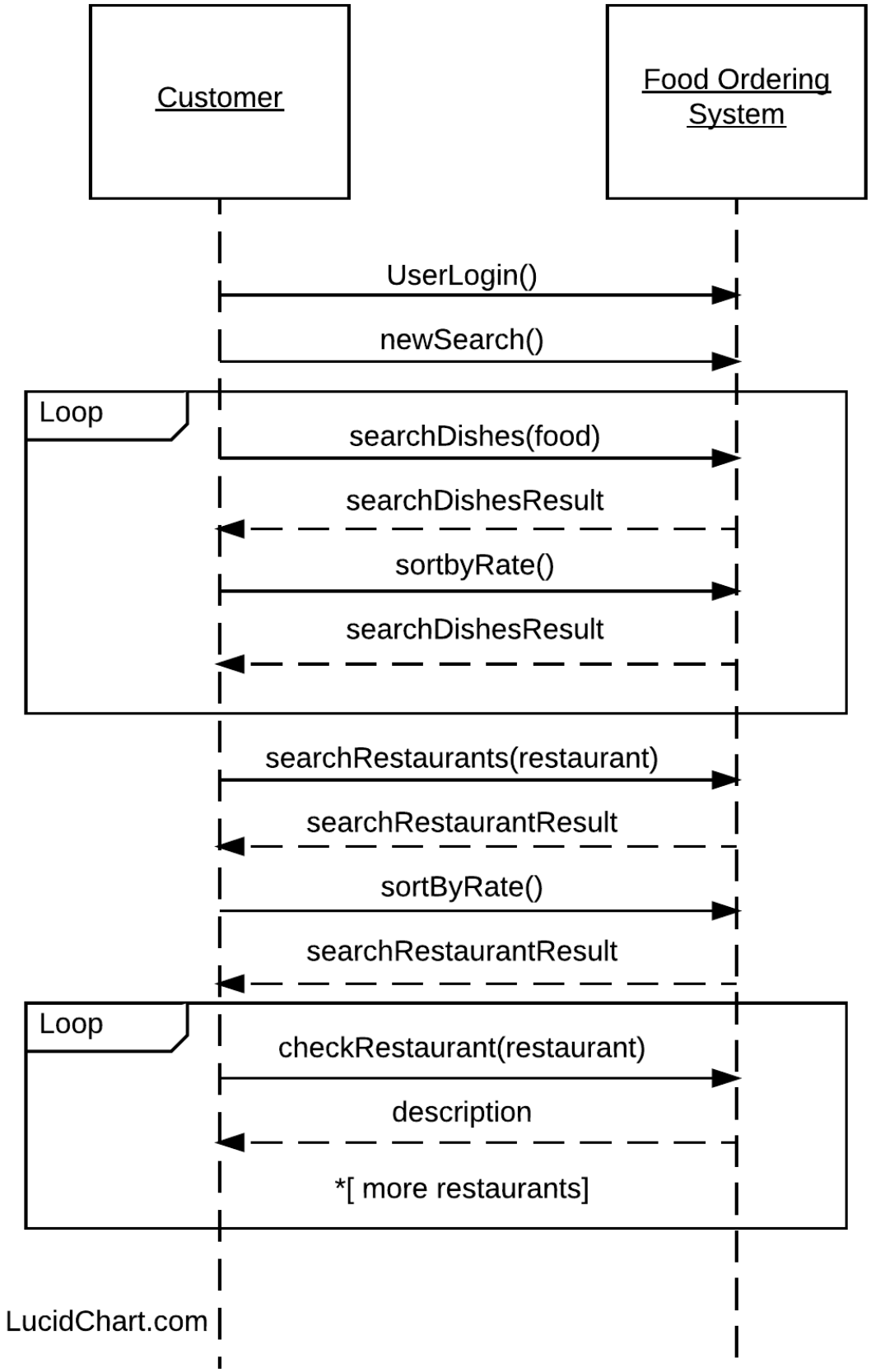


Price Sort System Sequence Diagram

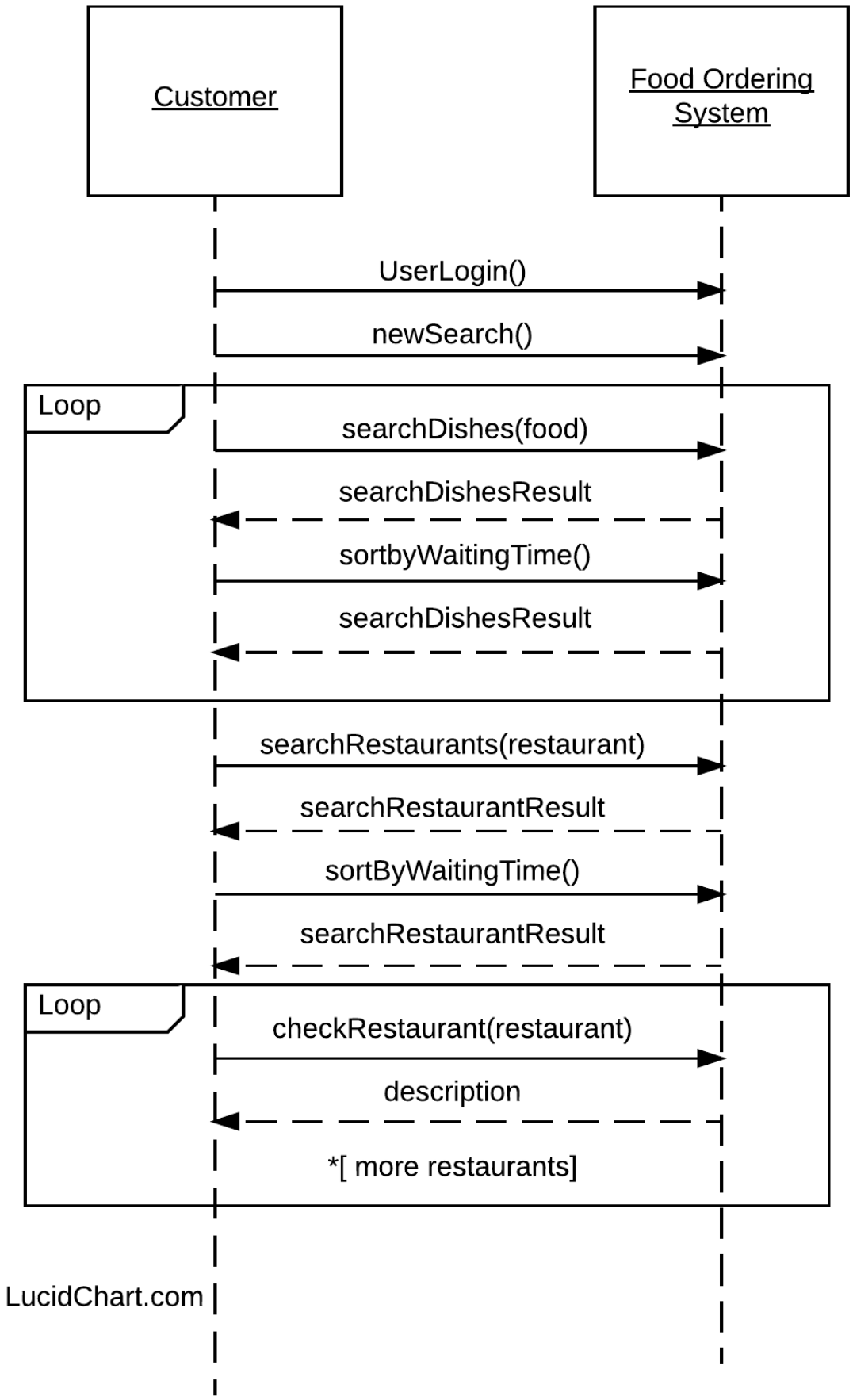


LucidChart.com

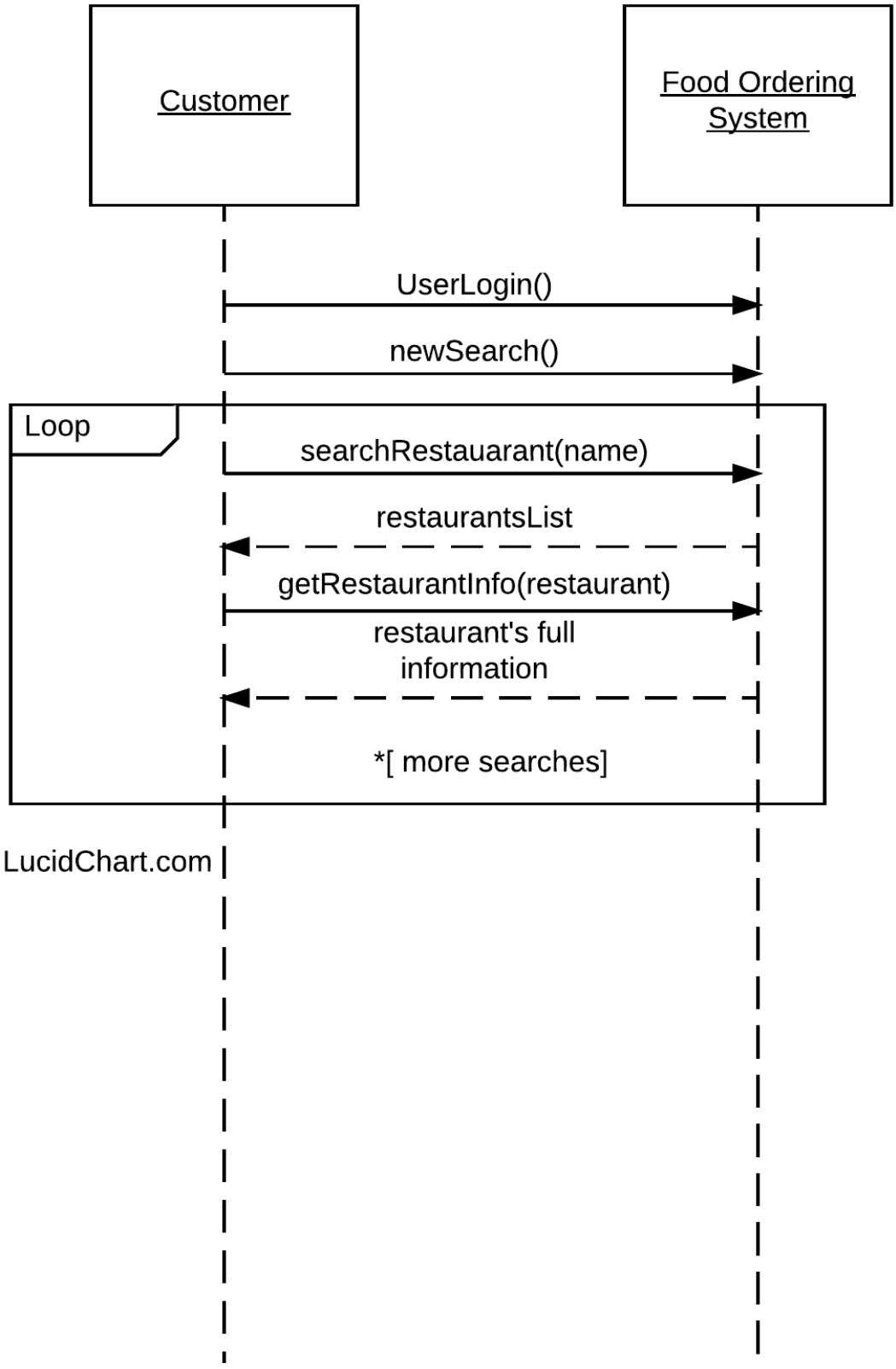
Ratings Sort System Sequence Diagram



Waiting Time Sort System
Sequence Diagram



Search Restaurant System
Sequence Diagram



2.8 Operation Contracts (3 marks)

Operation:	confirmEmail()
Cross References:	Uses Case: Account Creation
Pre conditions:	NIL
Post conditions:	-An email has been sent to the new user -User can use information in the email to confirm account creation

Operation:	customerReg(**firstName**: String, **lastName**: String, **email**: userEmail**, phoneNum**: userPhoneNumber, **location**: userLocation)
Cross References:	Use Case: Account Creation
Pre conditions:	NIL
Post conditions:	-Customer c was created (instance creation) -c was associated with Users (association formed) -A confirmEmail() operation created

Operation:	validateSearch(**queryedItem**: string)
Cross References:	Use Cases: Restaurant Search – Check's whether customer's search is correctly entered
Pre conditions:	NIL
Post conditions:	-Message returned that search was successful. Food Ordering System application will proceed to query the database.

Operation:	searchDatabase(**queryedItem**: string)
Cross References:	Use Cases: Restaurant Search – Searches the FOS Database queried restaurant.
Pre conditions	NIL
Post conditions:	Message returned that search was successful (or unsuccessful). -The database will send the restaurant's information to the application. The application will format the results in an easy to use UI.

Operation:	restaurantReg(**name**: string, **licence**: string, **open_time**: string, **close_time**: string)
Cross References:	use case: Account Creation
Pre conditions:	NIL
Post conditions:	-Restaurant r was created (instance creation) -r was associated with Users (association formed)

Operation:	modify_information()
Cross References:	uses cases: User Account Creation and Log In
Pre conditions:	- User is logged into their account
Post conditions:	- User can go through all the personal information of themselves - User can change their password in this operation - User can change their location in this operation - User can change their personal information in this operation

Operation:	selectFood()
Cross References:	Uses Cases: Placing an Order
Pre conditions:	- search instance s exists
Post conditions	- A order item instance oi was created - oi was associated with the current Search based on searchSpecification match

Operation:	checkout(**order**)
Cross References:	Uses Cases: Placing an Order
Pre conditions:	- Customer has selected food
Post conditions:	- An Order instance o was created (instance creation) - o was associated with the menu database (association formed) - order_total in o was initialized (attribute modification)

Operation:	addToOrder(**foodItem**)
Cross Reference:	uses cases: Placing Orders
Pre conditions:	NIL
Post conditions:	- A food item has been added the customer's chart. - An order price total been initialized or updated.

Operation:	makePayment()
Cross Reference:	Use Cases: Placing Orders
Pre conditions:	- A customer has confirmed the payment options and price
Post conditions:	- An order has been processed - System will notify the restaurant to prepare the food and transfer the money to the restaurant - The restaurant will prepare the food in the order after the transaction complete

Operation:	SortByPrice()
Cross References:	Uses Cases: Search for Price
Pre conditions:	- User is logged in
Post conditions:	- A list of applicable restaurants shown to users

Operation:	SortByWaitingTime()
Cross References:	Uses Cases: Search for Waiting Time
Pre conditions:	- User is logged in
Post conditions:	- A list of applicable restaurants shown to users

Operation:	setFoodItem(**name:** string, **price:** float)
Cross Reference:	Uses Cases: Placing an Order
Pre condition:	-Item must have valid input values
Post condition	-A Food instance f was created (instance creation) -f was associated with Restaurant (association formed)

Operation:	sign_in(**username:** string, **password:** string)
Cross Reference:	Uses Cases: Placing an Order, Search for Waiting Time, Search for Price, Search for Rating, Search for
Pre condition:	User exists in the database
Post condition:	User is logged in if information matches what is in the database

2.9 Obtaining User Feedback (1 mark)

User feedback was obtained after Milestone 5 when much of primary functionality of the application was completed. The group of four users that were involved with obtaining user feedback included Blazin_Seven's friends; The group included students who were studying computer science related fields and non-computer science related fields. Each user did not take more than 30 minutes to test our system. After demonstrating the functions of the program and asking the user to create an account, do searches, and complete a mock payment, we asked two primary questions:

1) Was the system easy to use and understand?
2) What changes can be made to make the application more appealing and intuitive?

The feedback that was given included changes to colour schemes, labels on buttons, and size of font on the system, adding functionalities to the account page, and positioning of search results. Though some recommendations were incorporated into our project, due to time constraints, much of the feedback obtained was not integrated.

3. Updated Design and Unit Testing (14 marks)

3.1 System Operations (1 mark)

SortByDistance() operation→ sort the restaurant/dish list by the distance between customer and restaurant

SortByRate() operation→sort the restaurant/dish list by the restaurant rate

SortByWaitimeTime() operation→sort the restaurant/dish list by the average waiting time of the restaurant

SortByPrice() operation→sort the dish list by the price of the dishes

NearbyRestaurants() operation→find the restaurants located at the same city with customer

Reg() operation → choose register as customer or restaurant

customerReg() operation → register as customer

RestaurantReg() operation → register as restaurant

createUserAccount(username, password) operation → once the user enters their information, an account is created and information is entered in the FOS database

sign_in() operation→ allowed user to log in the system

personal() operation→ allowed user modify their profile.

makePayment(total_price) operation →operation user pays for their order, with the price from the food items

selectFood() operation → selects a food item

addToOrder(foodItem) operation → puts a selected food item into the users open order

checkOut(order) operation → user starts the process of purchasing their order

emailVerification() operation → send user an email of a confirmation of account creation or of placing an order

changePassword() operation → allow users to change their passwords

changeLocation() operation → allow users to change their locations

newSearch() operation → starts a new search for the customer

searchDishes(food) operation → searches FOS database for the food item

searchRestaurants(restaurant) operation → searches FOS database for the restaurant

setFoodItem(name, price) operation → lets restaurants add food items to their menu

paymentOption() operation → Allows customers to choose which method to pay. The options are credit, debit, and cash.

notifyRestaurant(items, price) operation → notifies restaurants of an incoming order

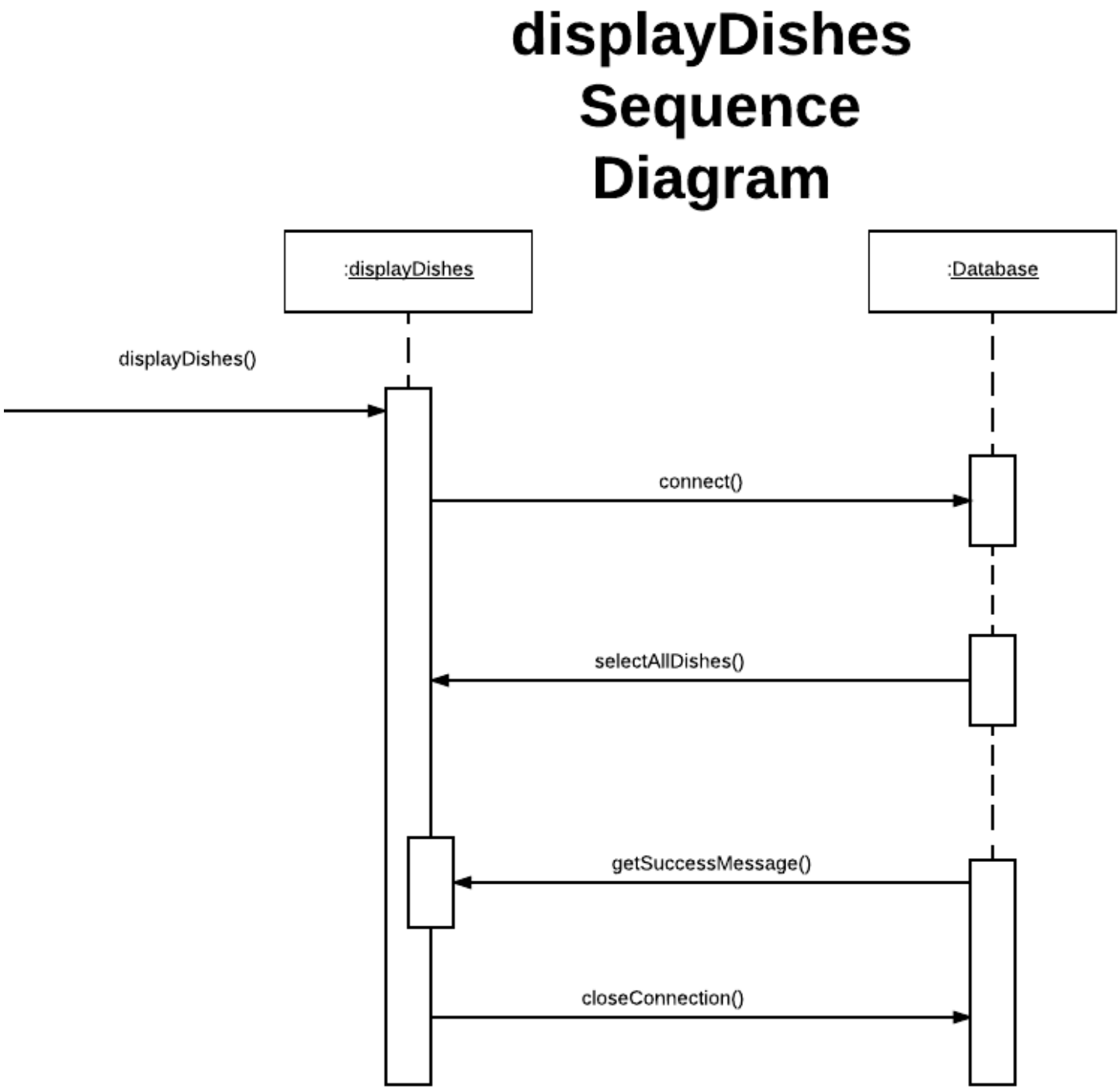
notifyCustomer(success) operation → notifies the customer that their order has been received by the restaurant

recieveFood() operation → customer notifies the system that order has been received

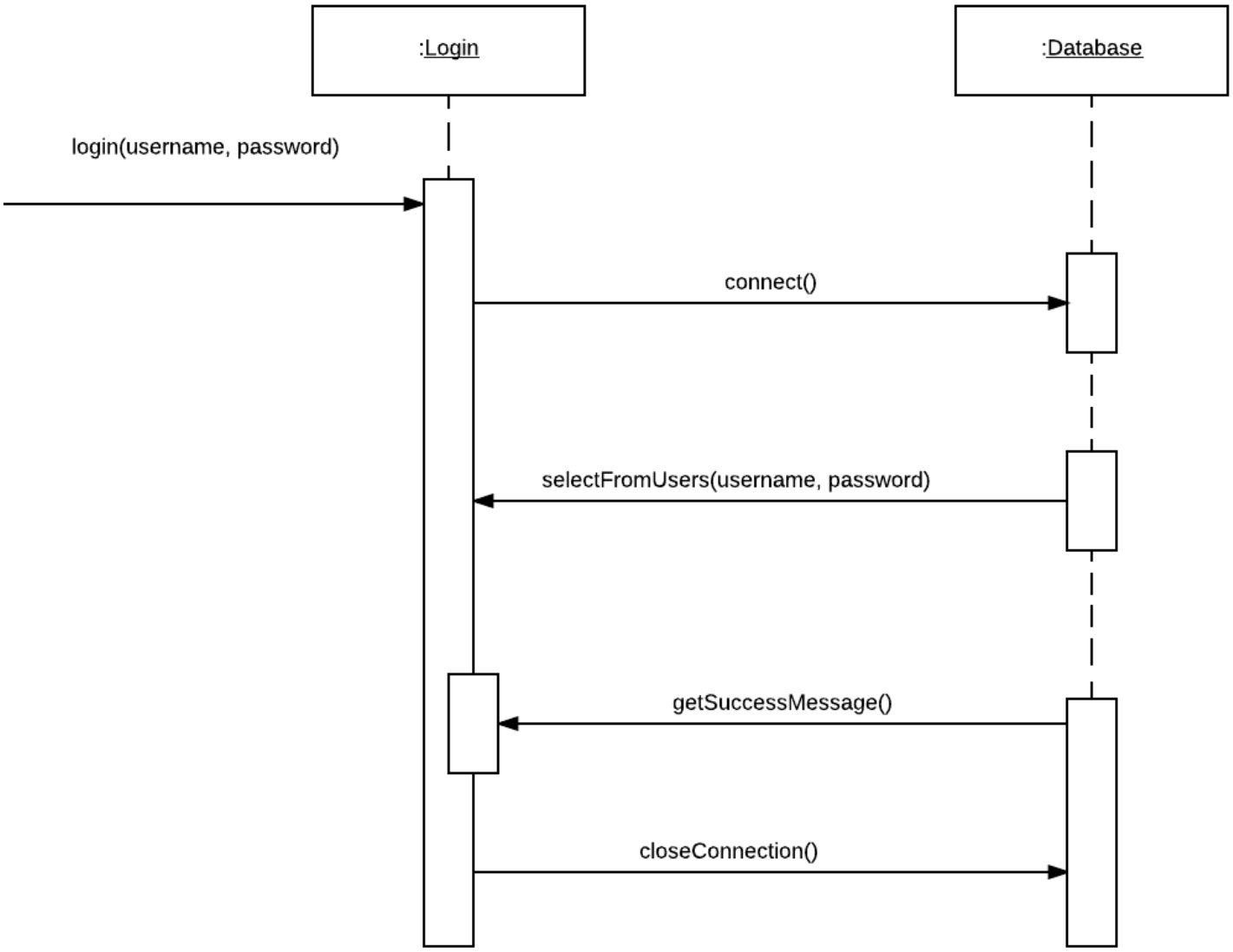
endOrder() operation → When an order has been received, the restaurant ends the order.

checkRestaurant(restaurant) operation → Choose a restaurant to get the full information of the restaurant (location, address, timings, ratings)

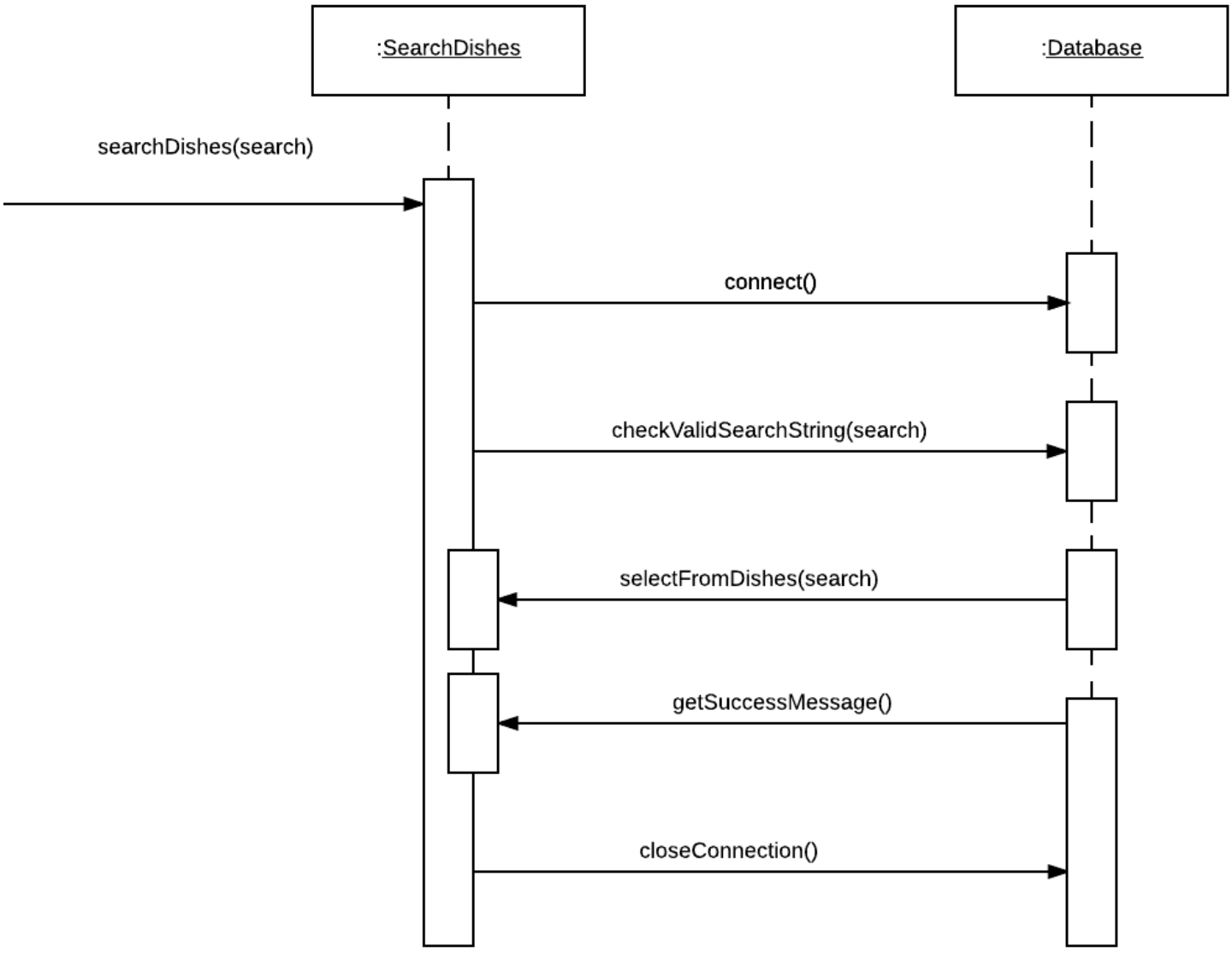
3.2 Sequence or Communication Diagrams with GRASP Patterns (2 marks)



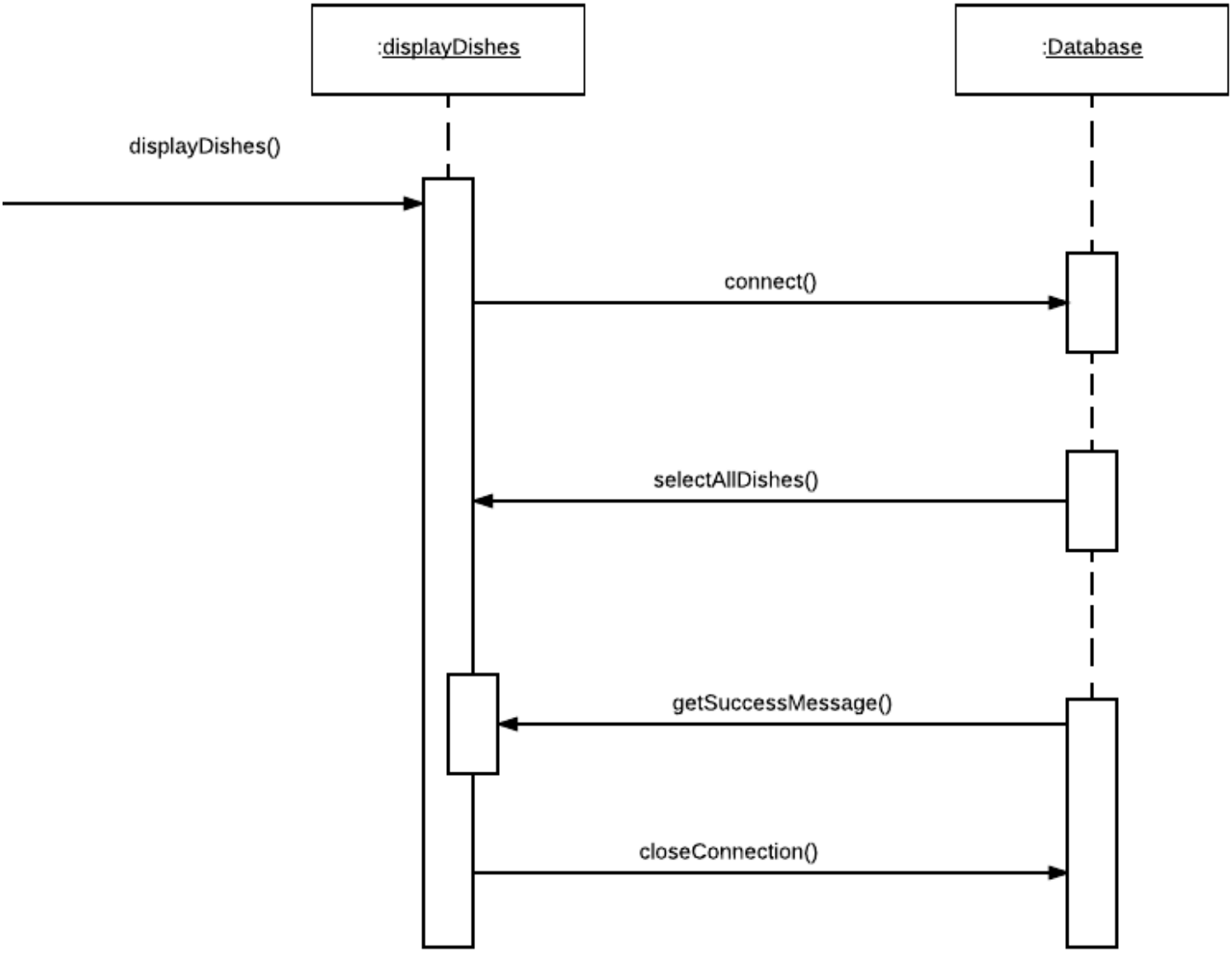
Login Sequence Diagram



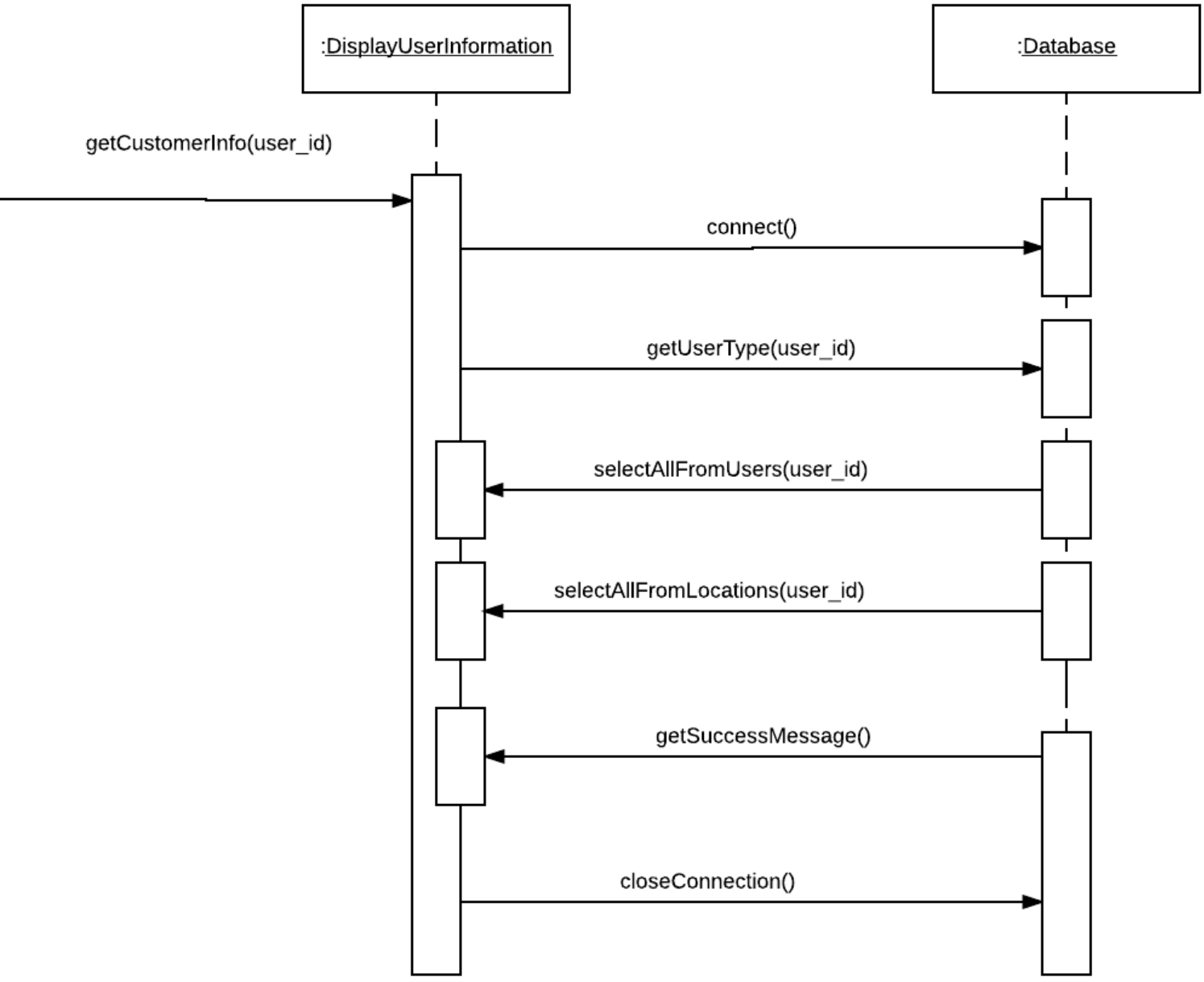
SearchDishes Sequence Diagram



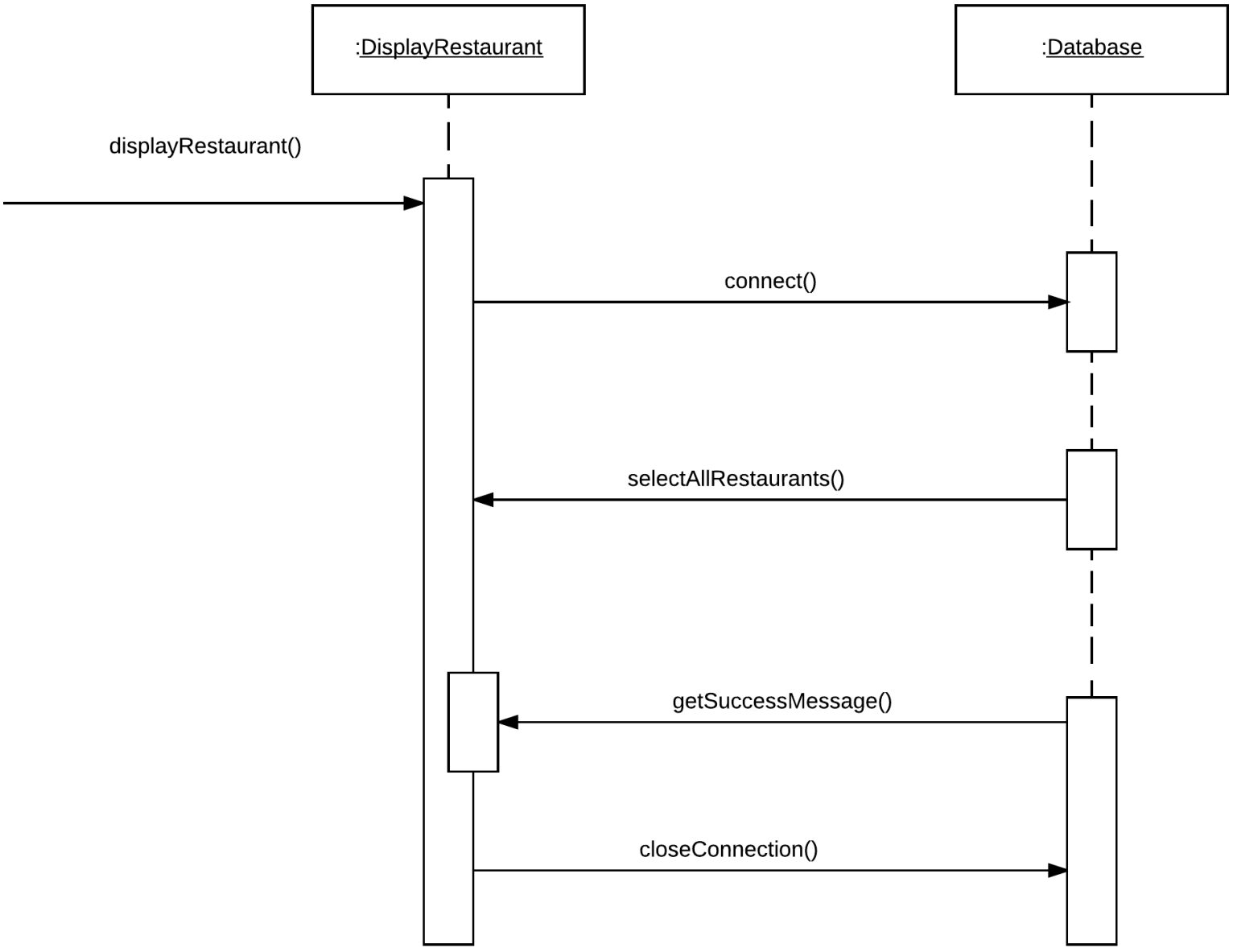
displayDishes Sequence Diagram



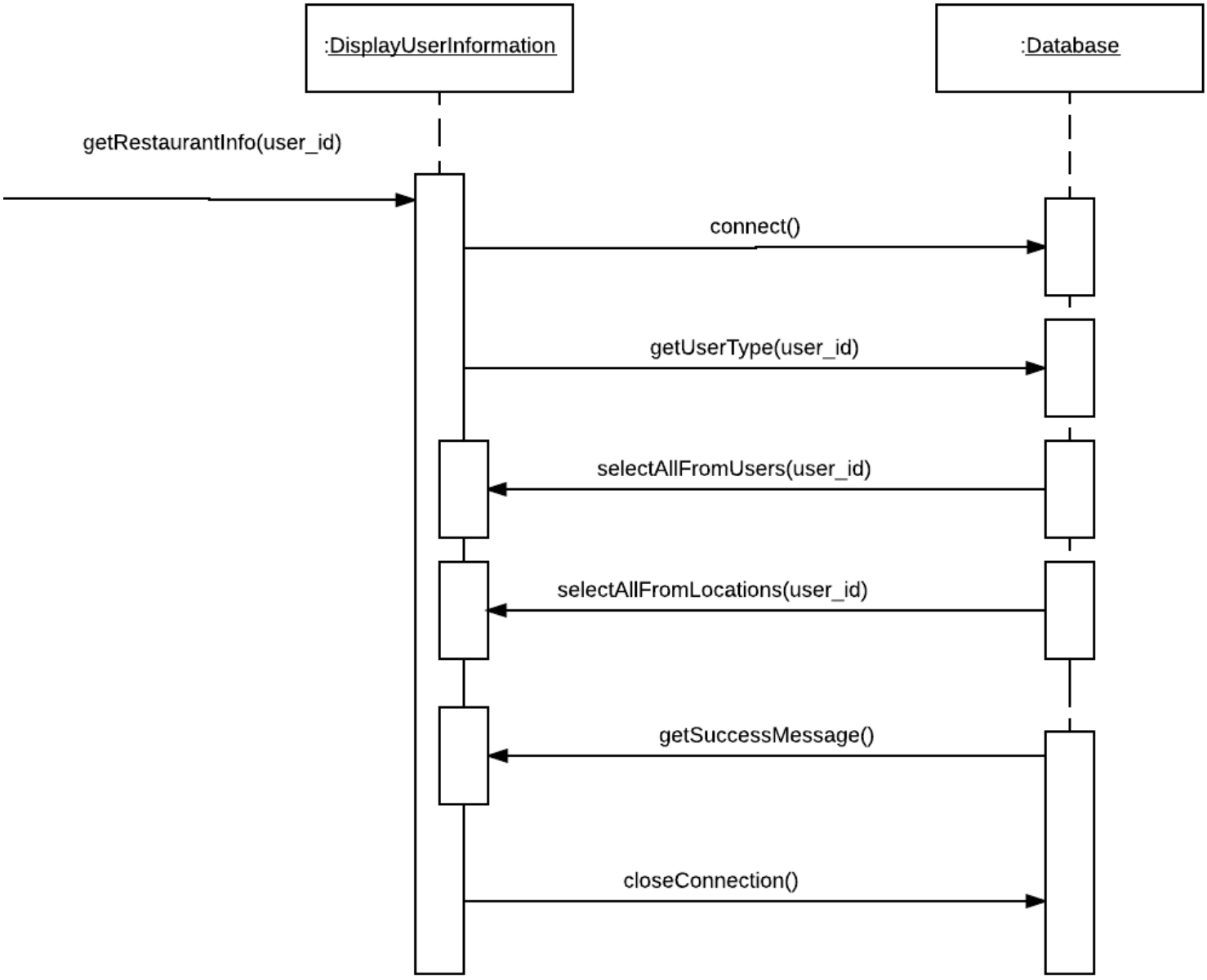
getCustomerInfo Sequence Diagram



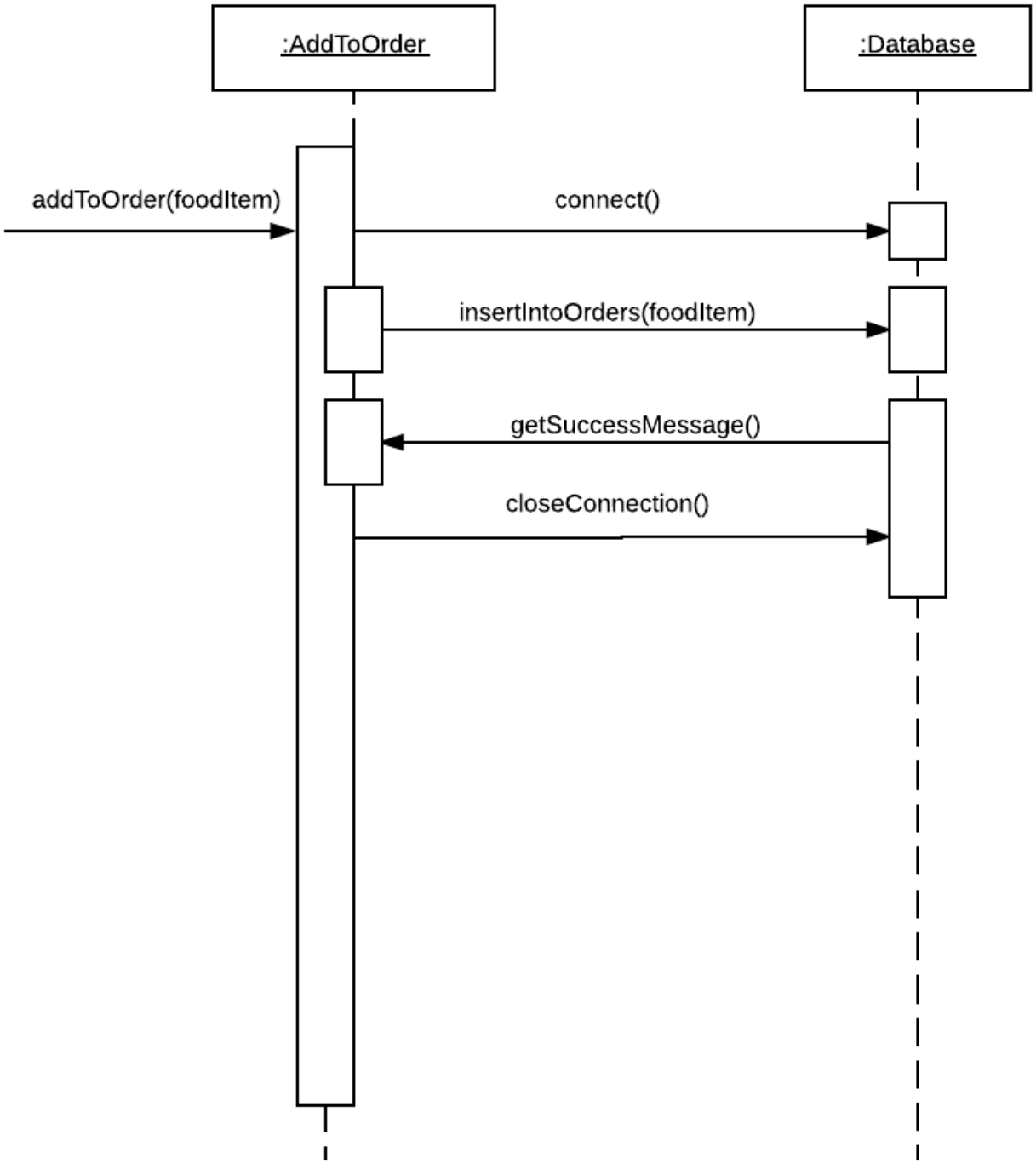
displayRestaurant Sequence Diagram



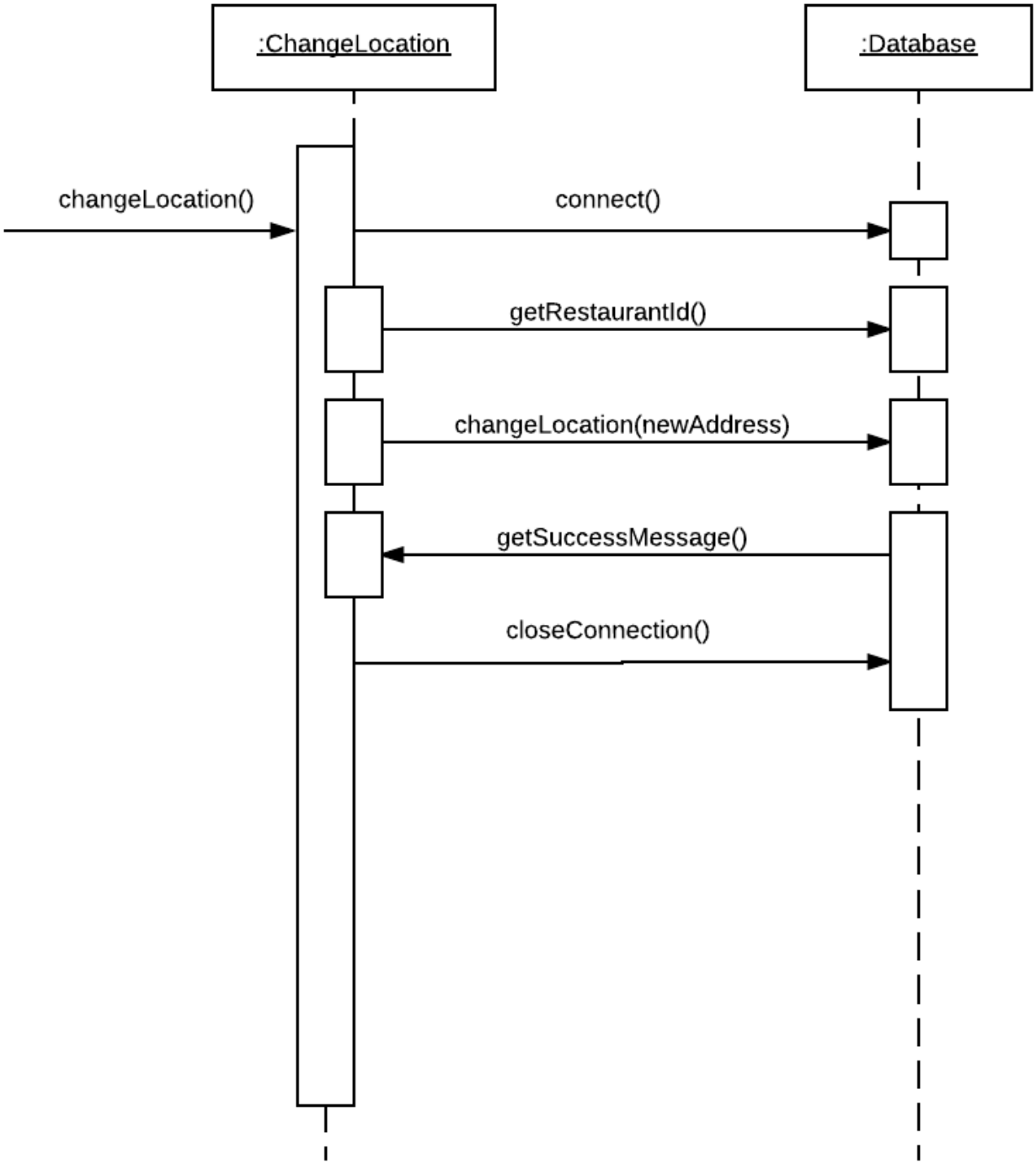
getRestaurantInfo Sequence Diagram



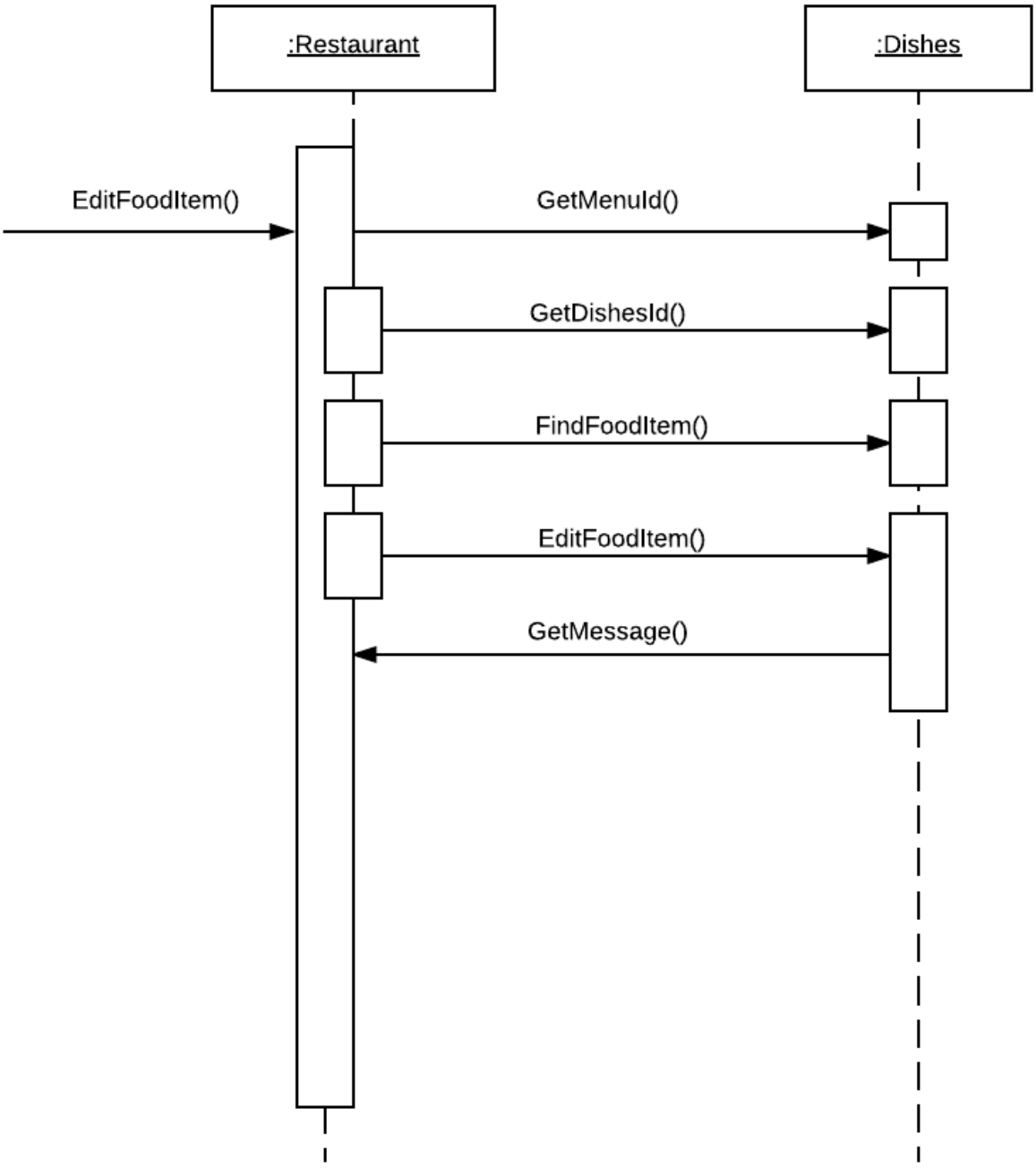
AddToOrder Sequence Diagram



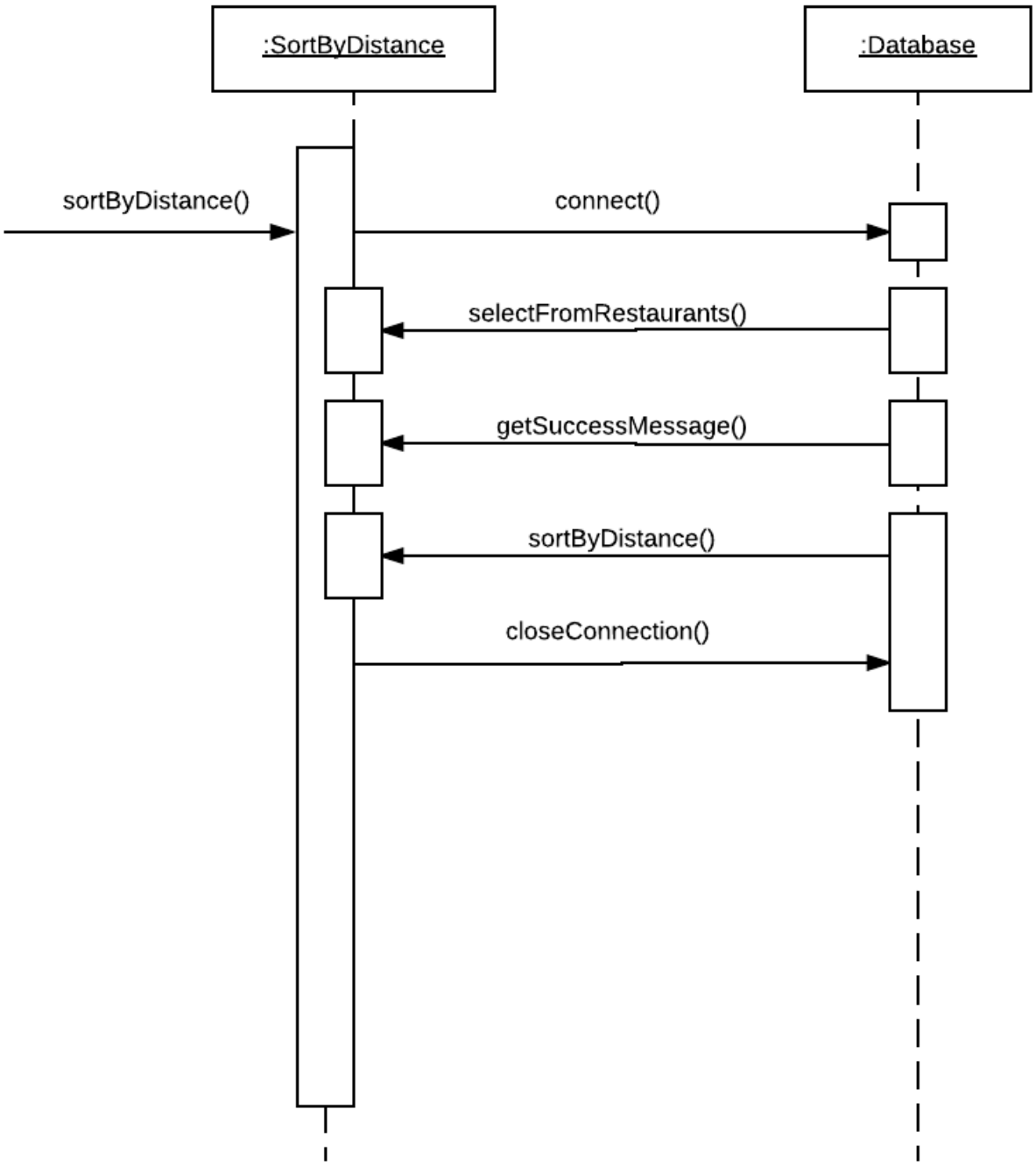
ChangeLocation Sequence Diagram



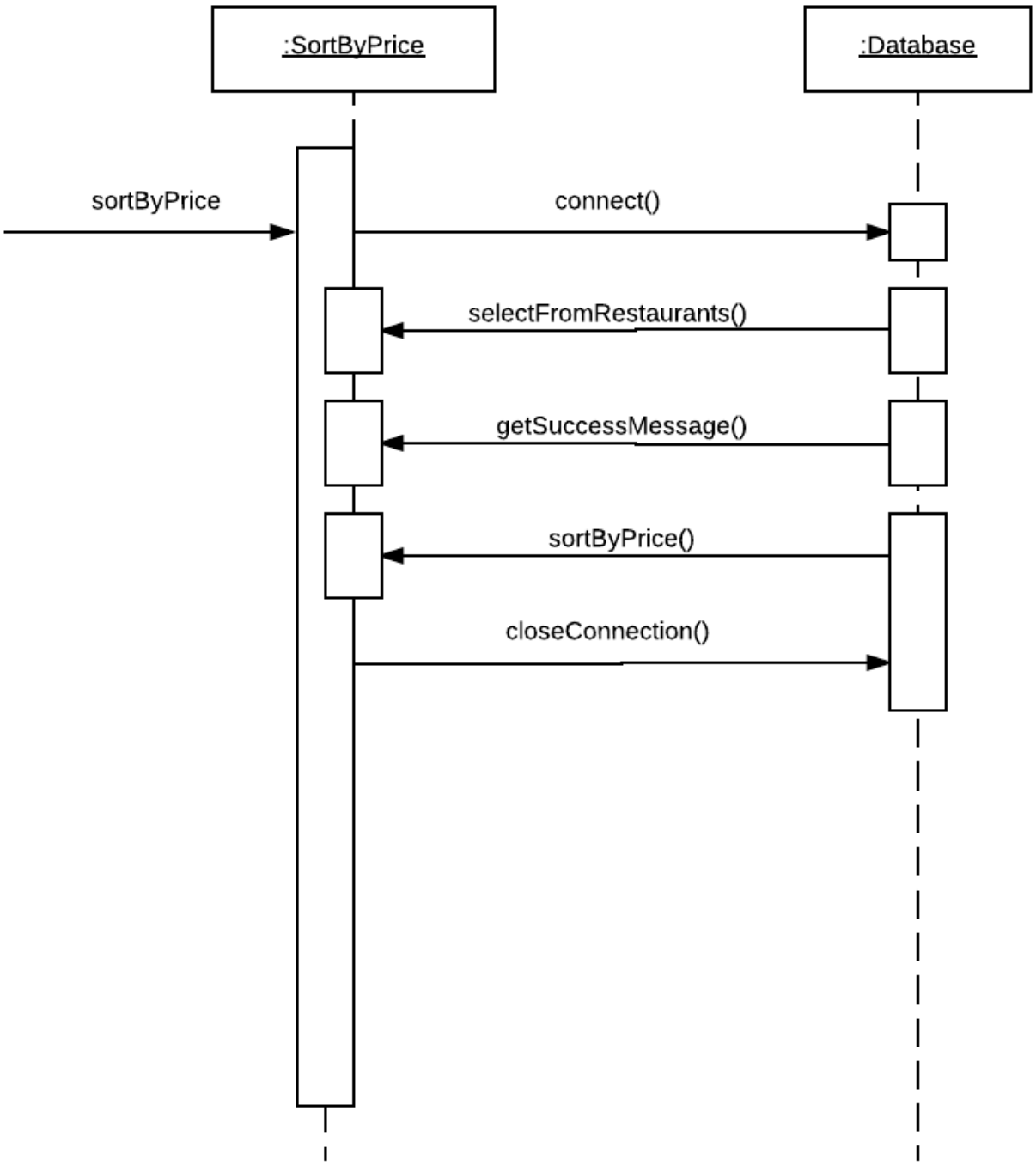
EditFoodItem Sequence Diagram



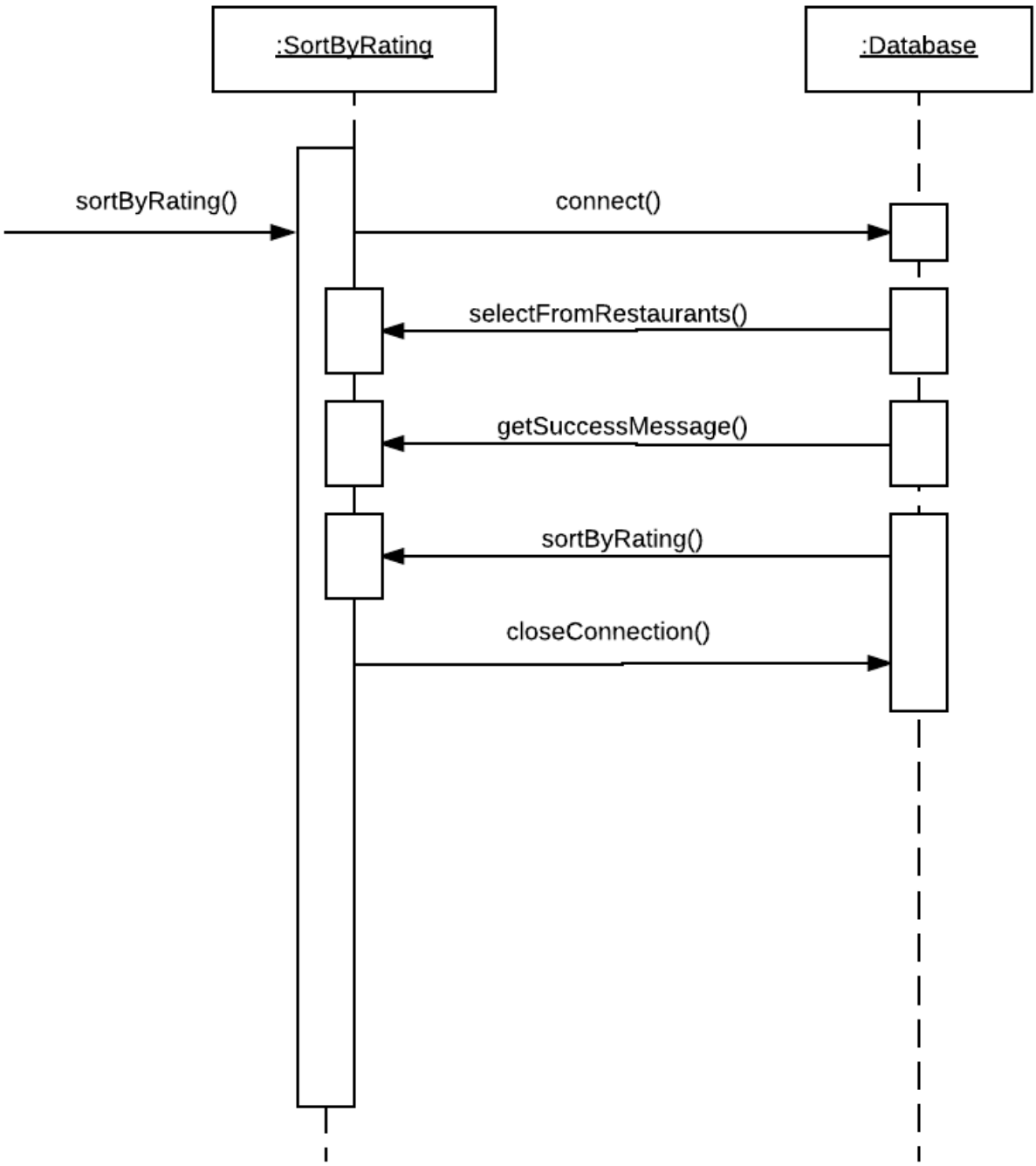
SortByDistance Sequence Diagram



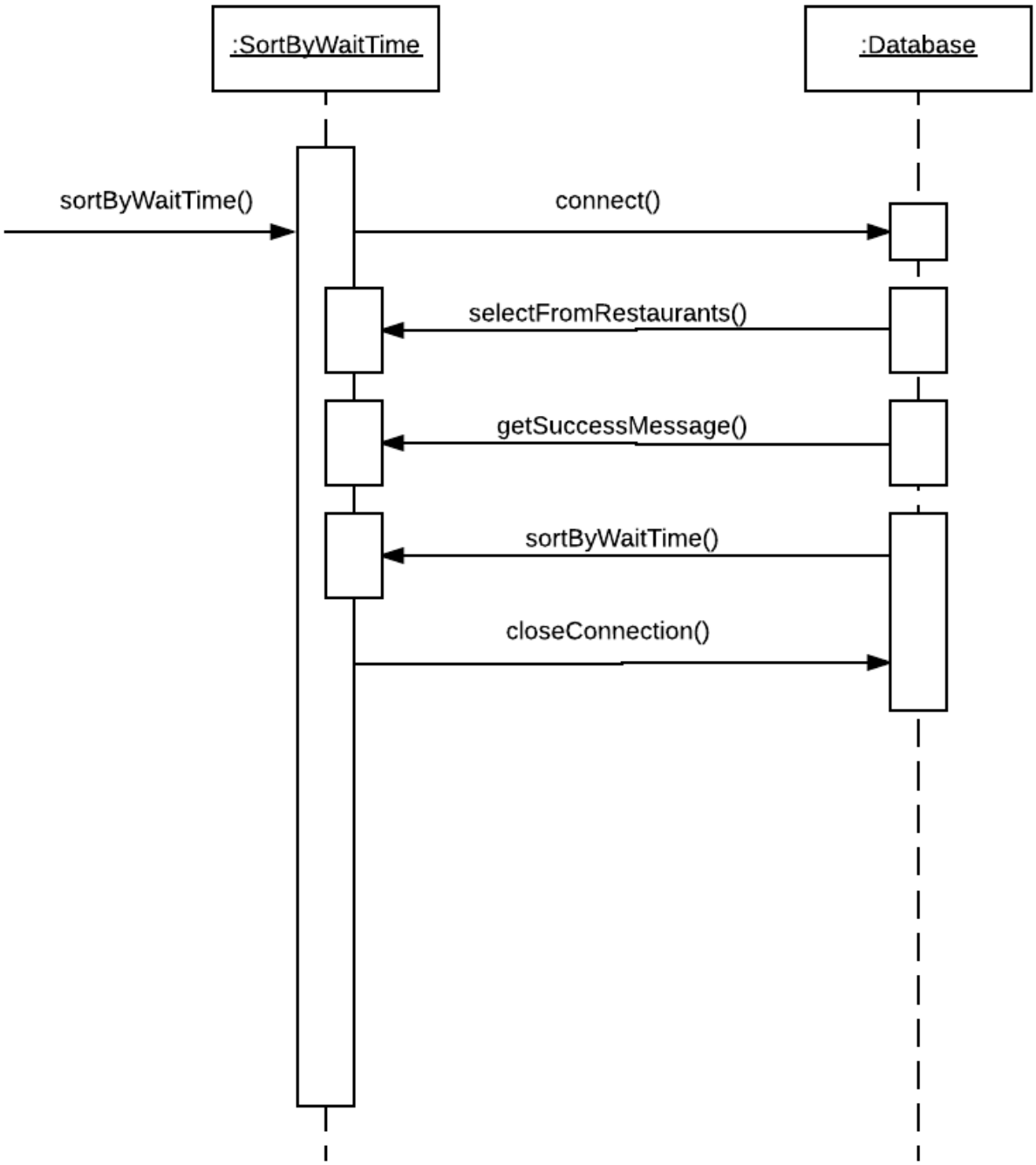
SortByPrice Sequence Diagram



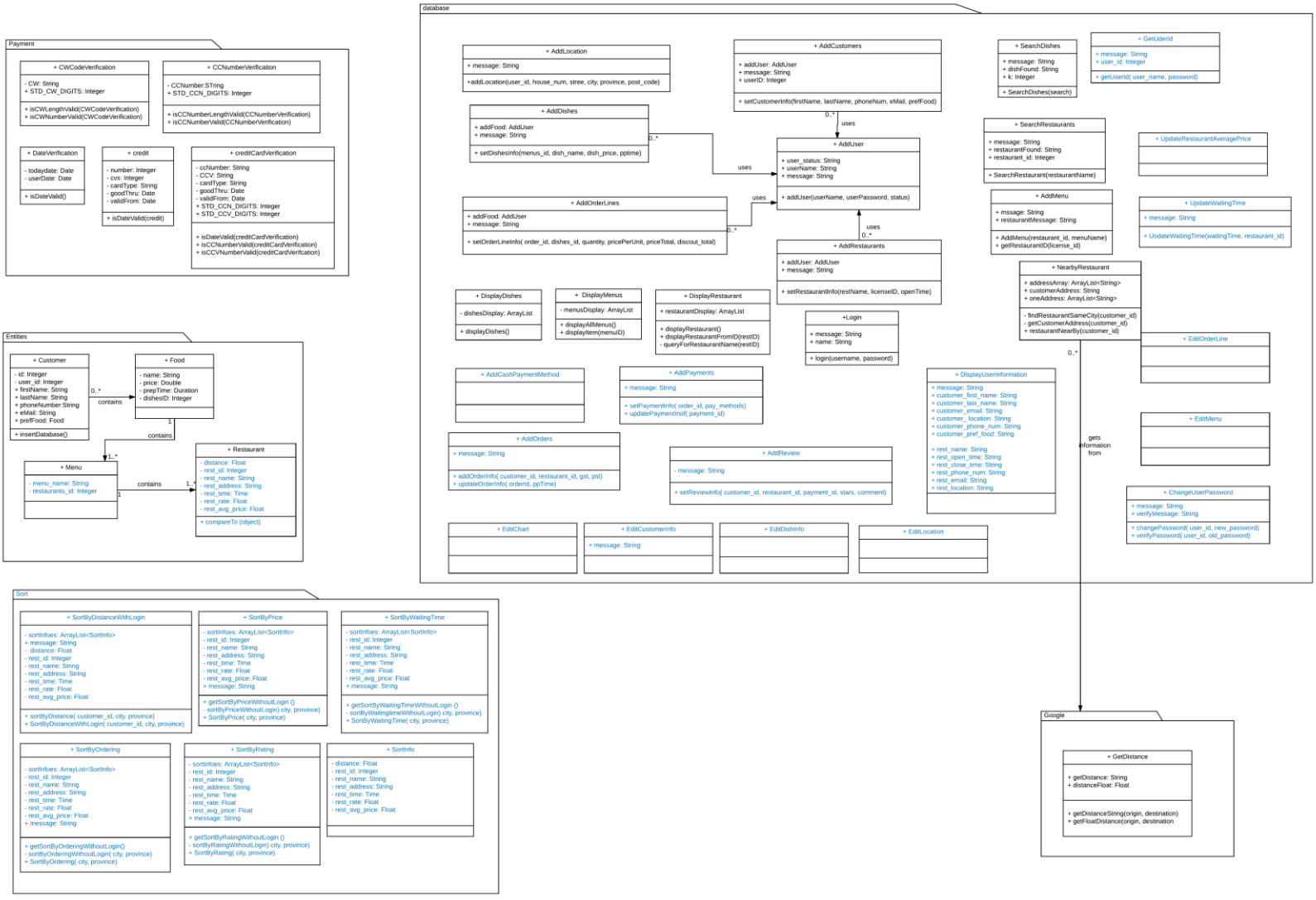
SortByRating Sequence Diagram



SortByWaitingTime Sequence Diagram



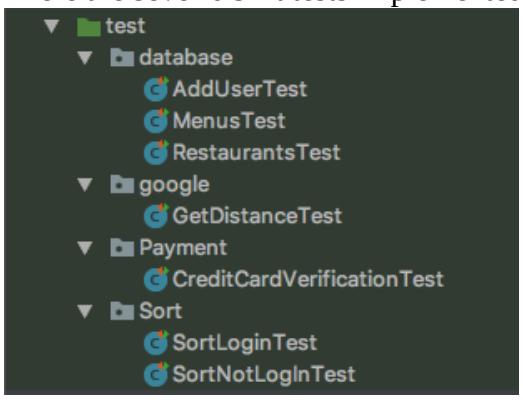
3.3 Class Diagram (3 marks)



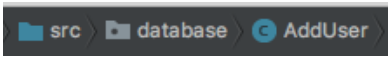
** Blue font are new classes and methods
** For a clearer image, please see attached class diagram in file

3.4 Unit Testing (8 marks)

All the test codes are in the "test" folder.
There are seven JUnit tests implemented into the project:



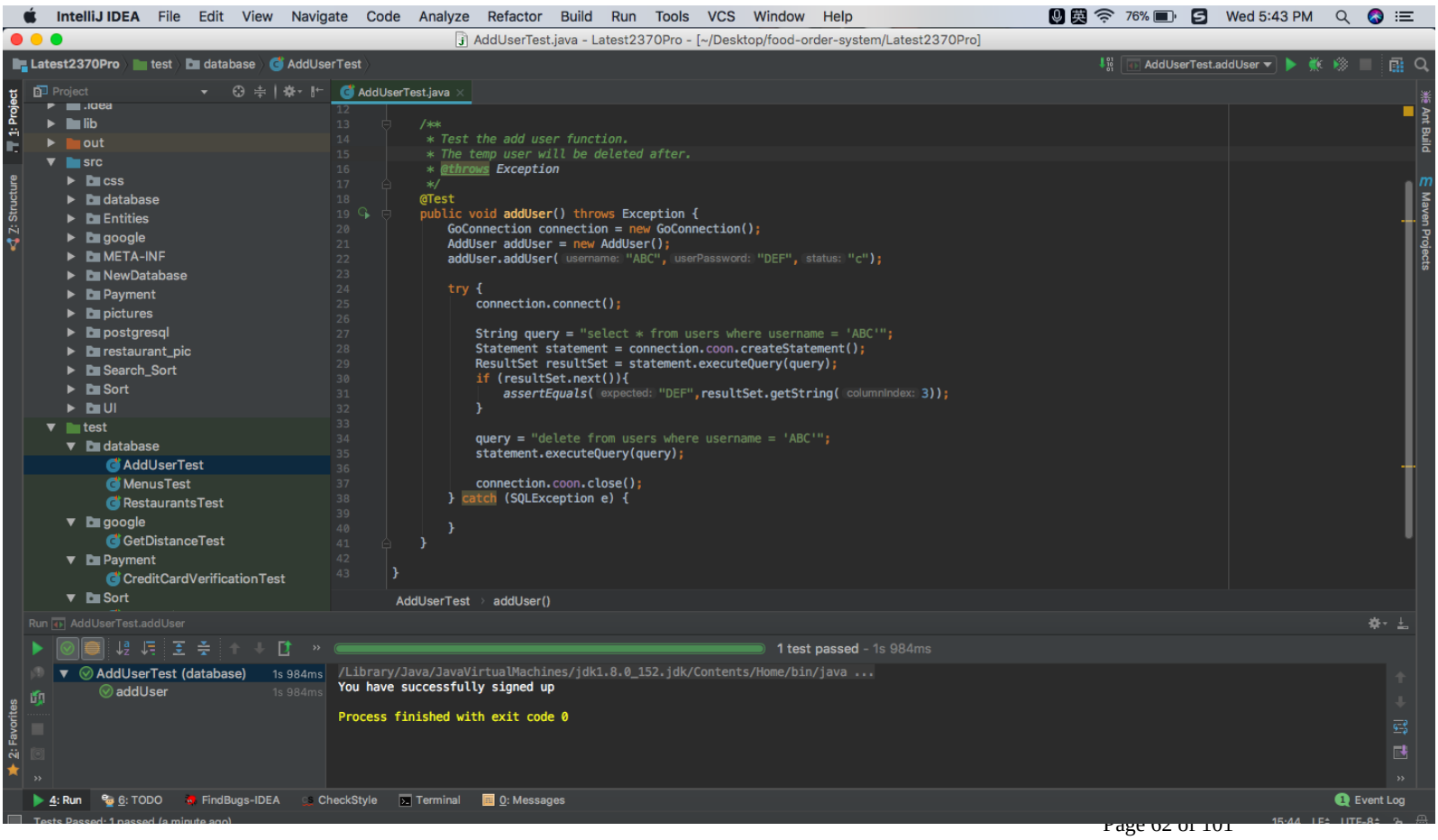
1) **AddUserTest**:

Test object (source code): 

Line 31 – 63, method "addUser"

How: apply JUnit to method "addUser"

Testing data



Usability:
Make tester knows whether the method “addUser” works correctly or not.

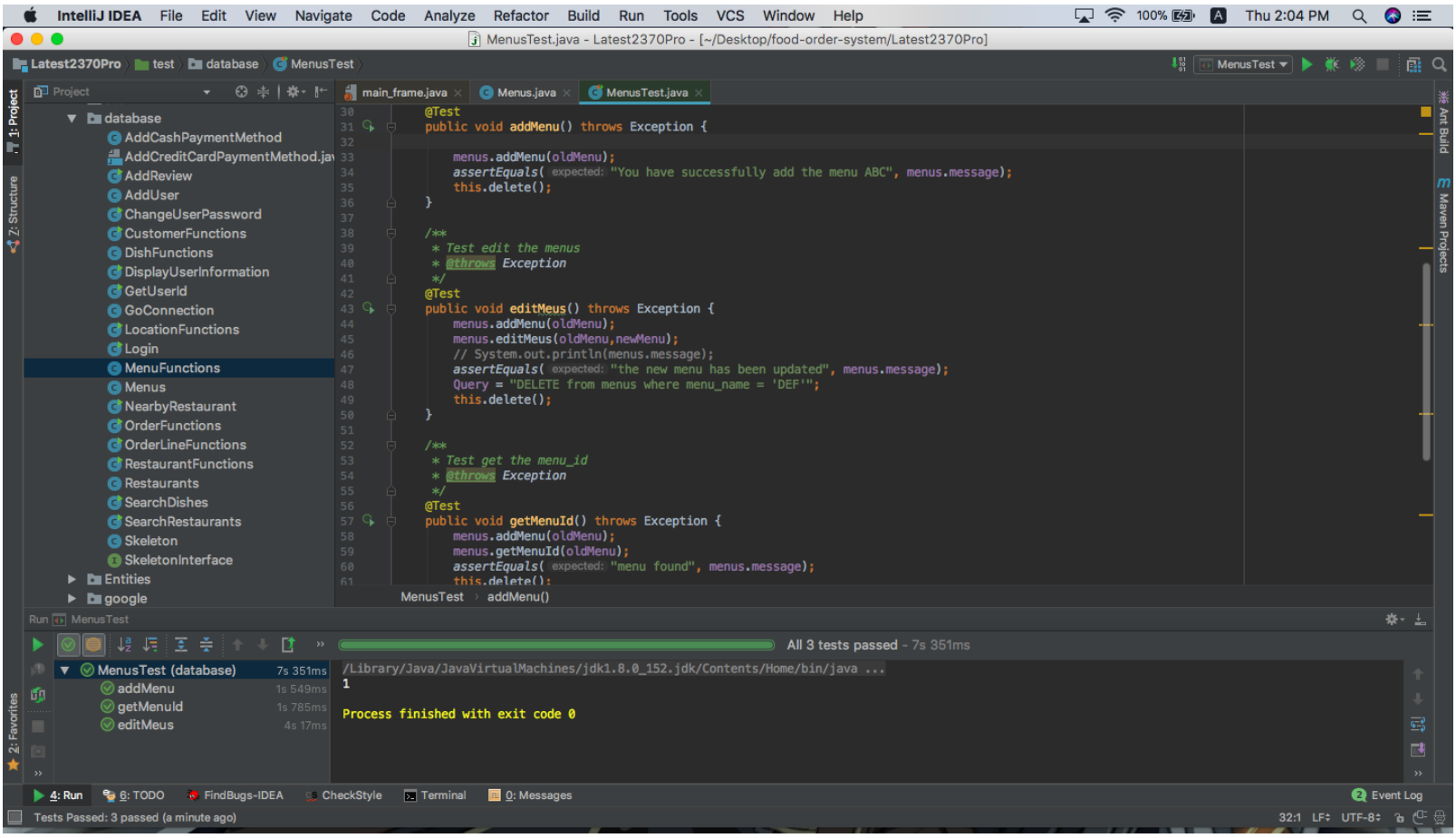
2) **MenusTest**

Test object (Source code): src > database > Menus

Line 107 - 119, method “addMenu”; Line 121-132, method “editMenus”; Line 134 - 144, method “getMenuId”;

How: apply JUnit to method “addMenu”, “editMenus” and “getMenuId”

Testing data:



Usability:
Make tester knows whether the three methods “addMenu”, “editMenus” and “getMenuId” work correctly or not.

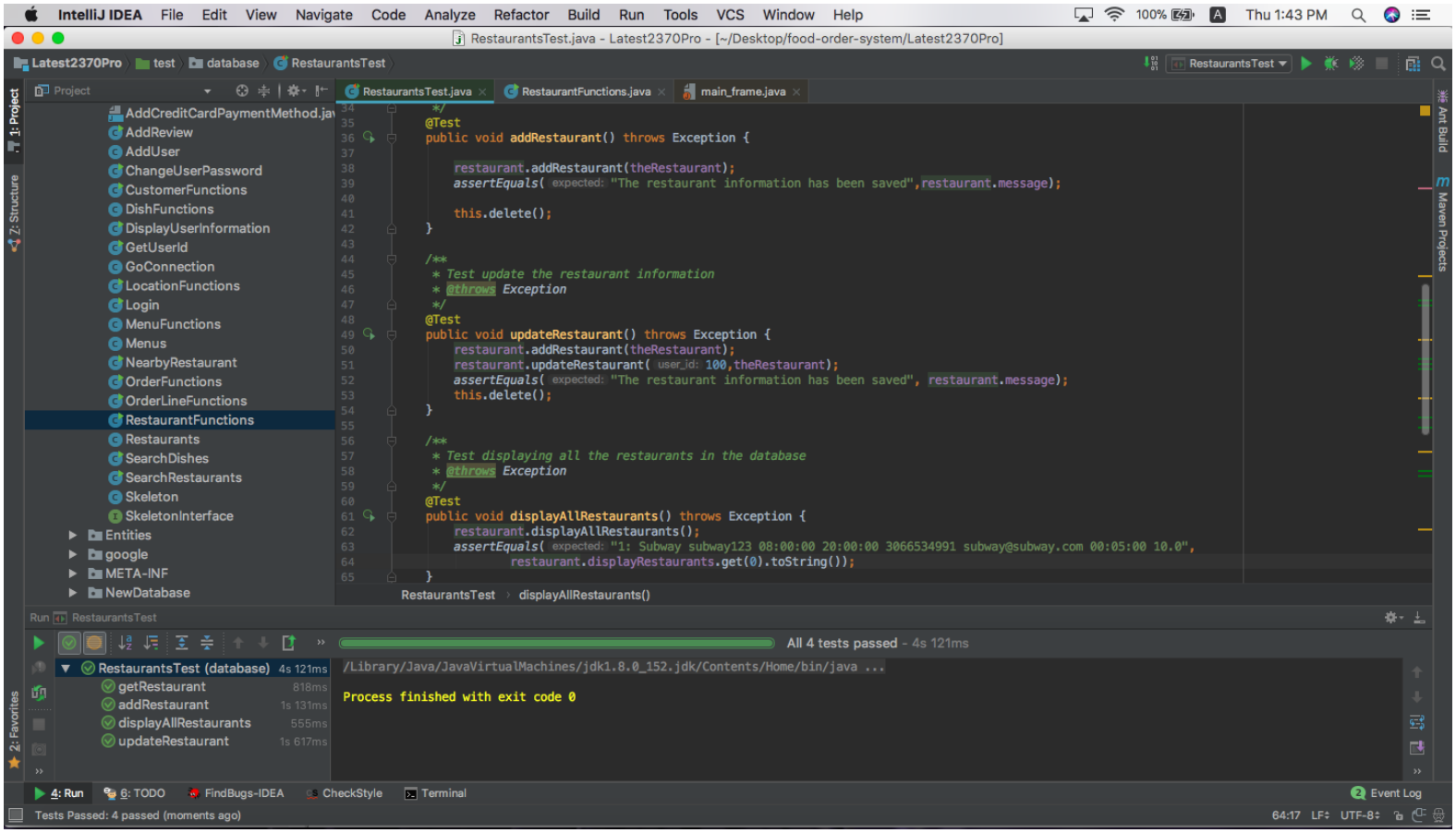
3) **RestaurantsTest**

Test object (Source code): src > database > Restaurants

Line 197-205, method “addRestaurant”; Line 207-225, method “editRestaurant”; Line 229-239, method “displayAllRestaurants”; Line 241-251, method “getRestaurant”

How: apply JUnit to method “addRestaurant”, “editRestaurant”, “displayAllRestaurants” and “getRestaurant”

Testing data:



Usability:

Make tester knows whether the three methods “addRestaurant”, “editRestaurant”, “displayAllRestaurants” and “getRestaurant” work correctly or not.

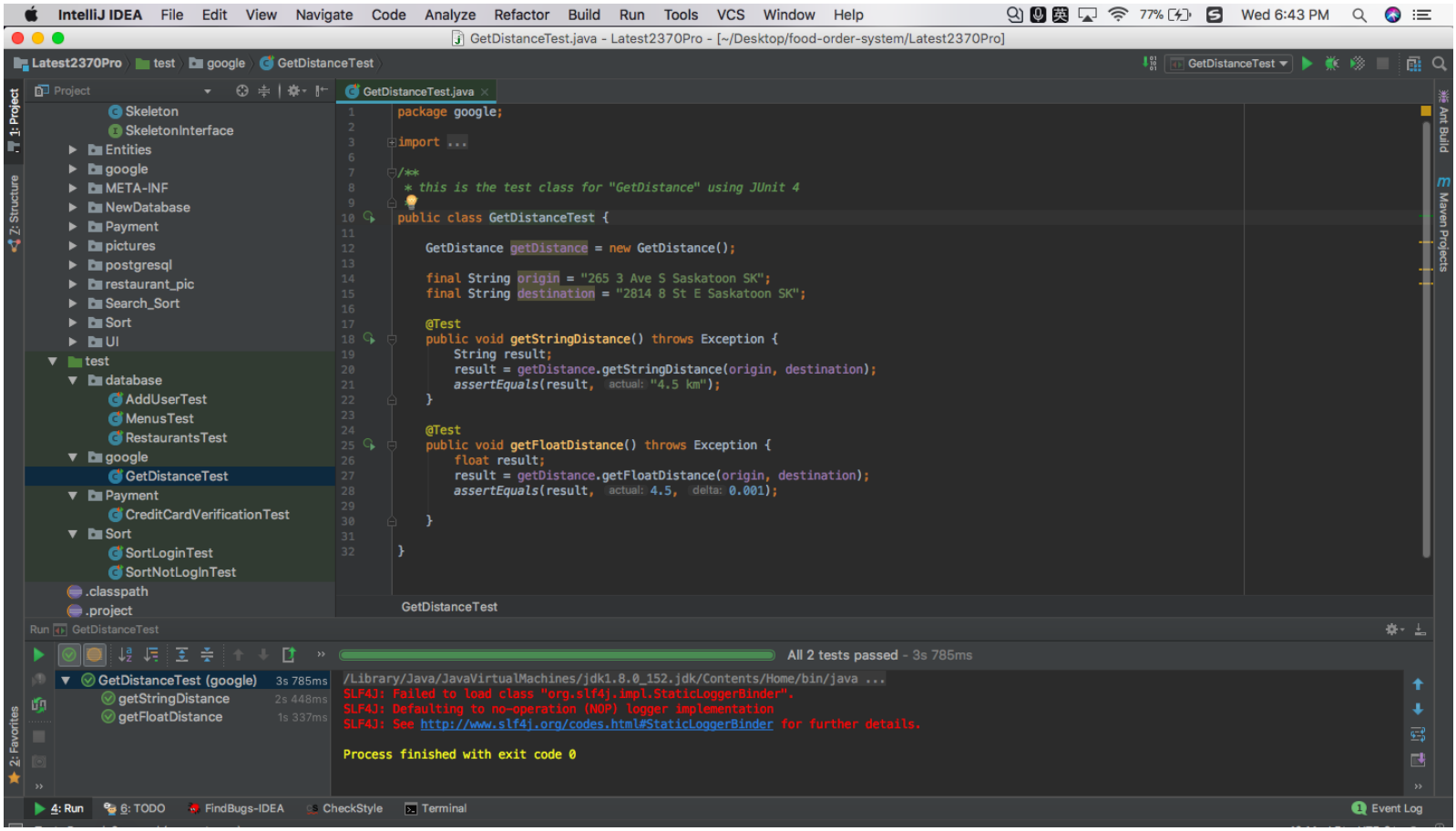
4) **GetDistanceTest**

Test object (Source code): src > google > GetDistance

Line 41 – 62, method “getStringDistance”; Line 64 – 80, method “getFloatDistance”

How: apply JUnit to method “getStringDistance” and “getFloatDistance”

Testing data:



Usability:
Make tester knows whether the method “getStringDistance” and method “getFloatDistance” work correctly or not.

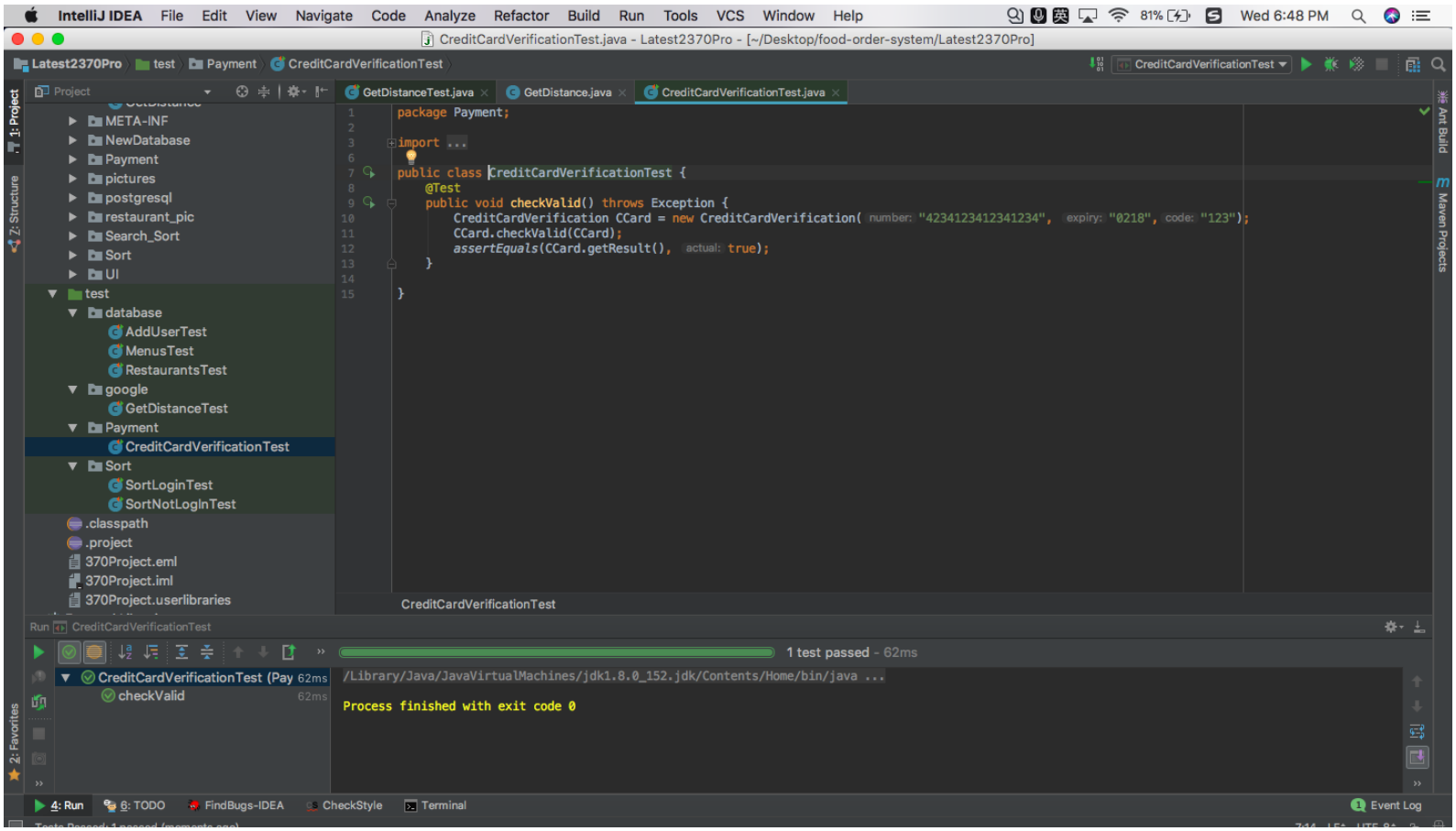
5) **CreditCardVerificationTest**

Test object (Source code): src > Payment > CreditCardVerification

Line 60-79, method “checkValid”.

How: apply JUnit to method “checkValid”

Testing data:



Usability:
Make tester knows whether the method “checkValid” works or not

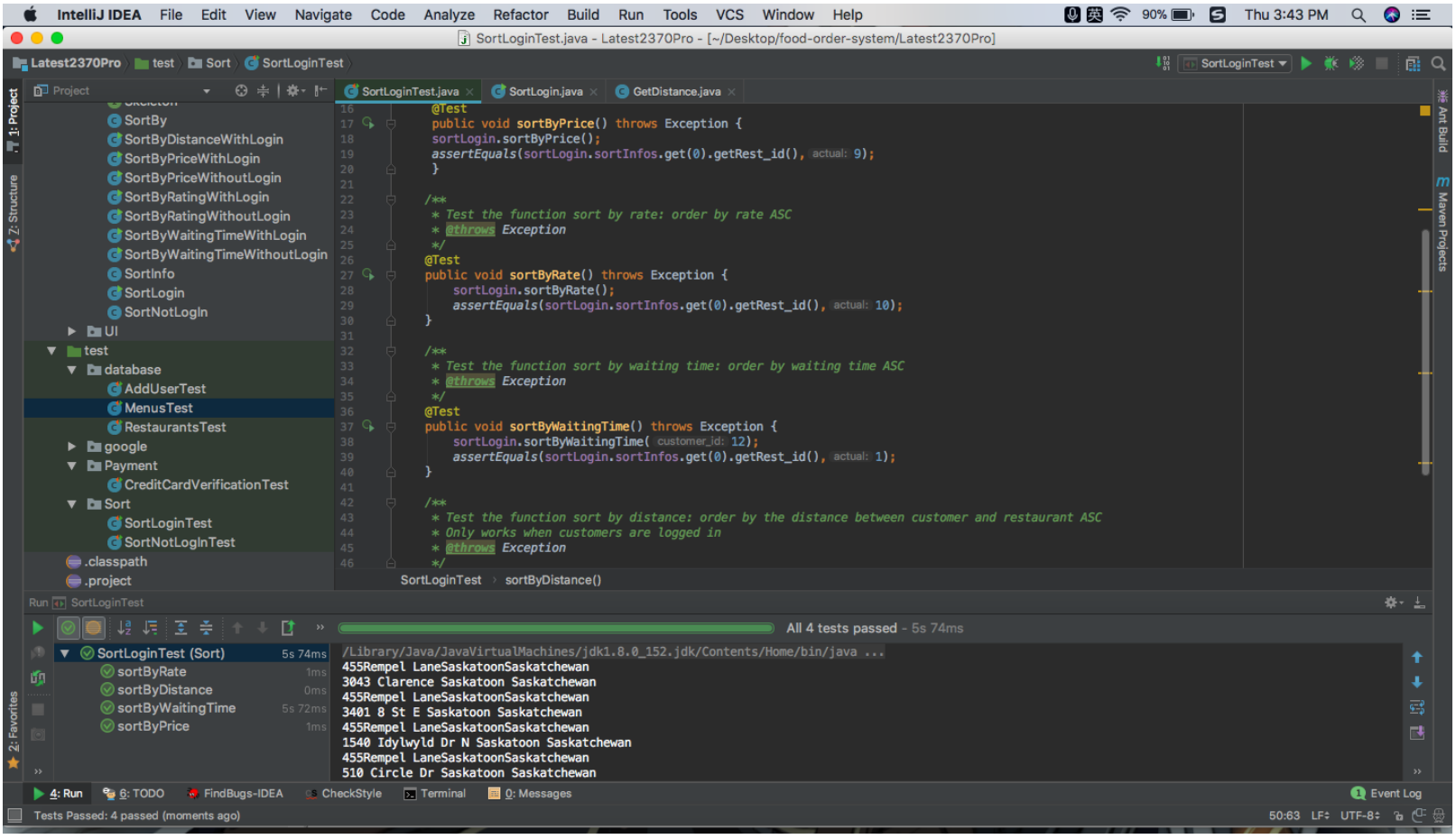
6) **SortLoginTest**

Test object (Source code): src > Sort > SortLogin

Line 110-113, method “sortByPrice”; Line 117-120, method “sortByRate”; Line 125-149, method “sortByWaitingTime”; Line 151-154, method “sortByDistance”

How: apply JUnit to method “sortByPrice”, “sortByRate”, “sortByWaitingTime” and “sortByDistance”

Testing data:



Usability:
Make tester knows whether the three methods “sortByPrice”, “sortByRate”, “sortByWaitingTime” and “sortByDistance” work correctly or not.

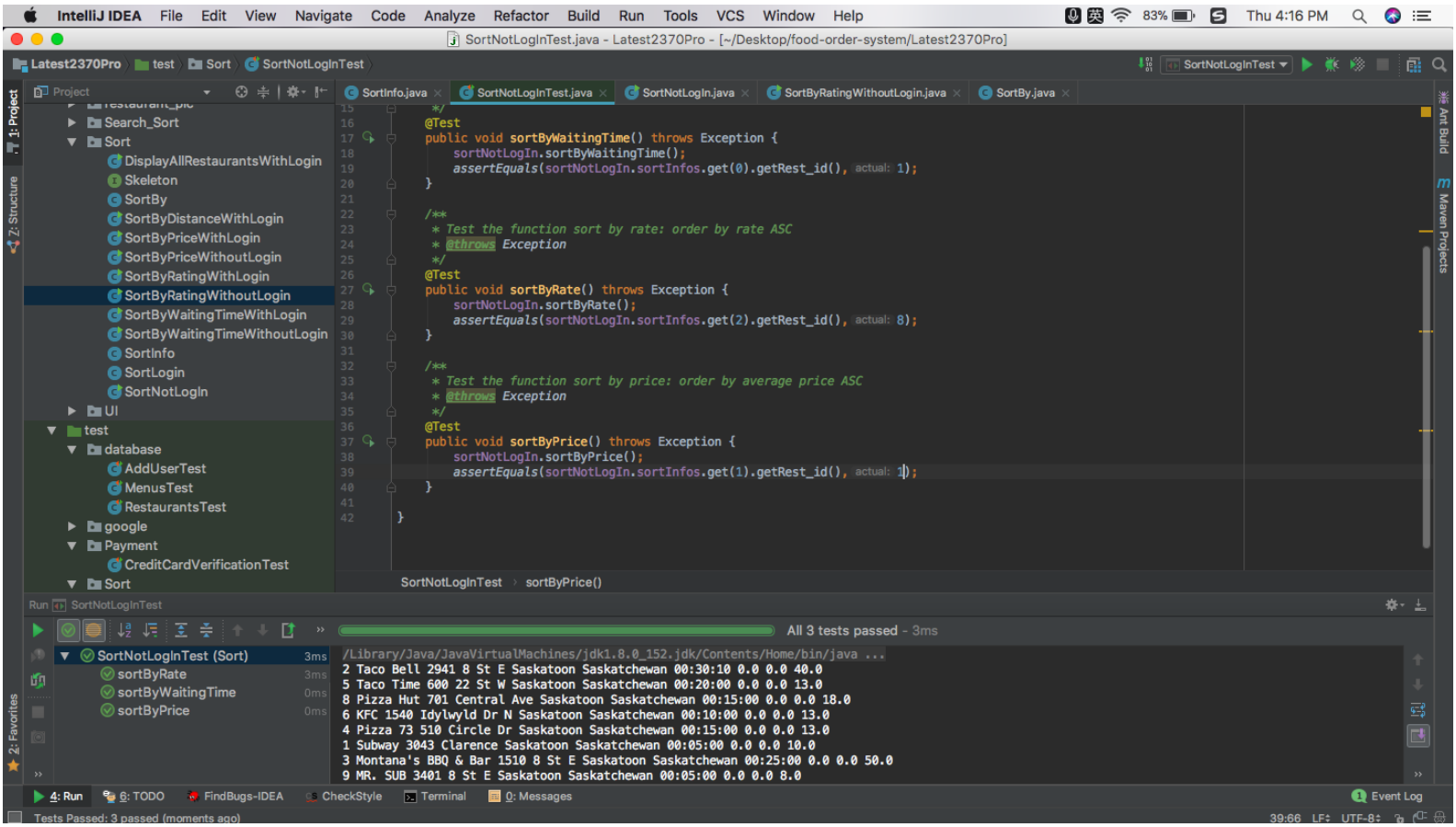
7) **SortNotLogInTest**

Test object (Source code): src > Sort > SortNotLogIn

Line 69-75, method “sortByPrice”; Line 62-65, method “sortByRate” and Line 55-58, method “sortByWaitingTime”

How: apply JUnit to method “sortByPrice”, “sortByRate” and “sortByWaitingTime”

Testing data:



Usability:
Make tester knows whether the three methods “sortByPrice”, “sortByRate” and “sortByWaitingTime” work correctly or not.

4. Reengineering (22 marks + 2 bonus marks)

4.1 Code Smells

Clone Pair/ Class ID	Root causes	Further comments
1	1.a(i)	Reusing existing code by copying and pasting with object been modified. Solution in final project: class no longer been used.
2	1.a(i)	Reusing existing code by copying and pasting with different SQL Query and minor modification. Solution in final project: abstract the clone part as a new function
3	1.a(i)	Reusing existing code by copying and pasting with different SQL Query and minor modification. Solution in final project: abstract the clone part and make it as a new parent class
4	1.a(i)	Reusing existing code by copying and pasting with no modification. Solution in final project: abstract the clone part as a new function
5	1.a(ii)	Reuse similar solutions: used in different situations with different input Query: one of them get String while the other one will get integer. Solution in final project: class no longer been used. New class will replace the old one.
6	1.a(i)	Reusing existing code by copying and pasting with different SQL Query only. Solution in final project: abstract the clone part as a new function.
7	1.a(i)	Reusing existing code by copying and pasting with different SQL Query and minor modification. Solution in final project: abstract the clone part as a new function
8	1.a(i)	Reusing existing code by copying and pasting with different SQL Query and minor modification. Solution in final project: abstract the clone part as a new function. Can be done with clone 7.
9 10 11	1.a(i)	Same as 7 and 8 and refactoring can be done when refactoring 7 and 8.
12	1.a(i)	Reusing existing code by copying and pasting with object been modified. Solution in final project: class no longer been used. New class will replace the old one.

13	1.a(ii)	Reuse similar solutions: used in different situations with different SQL situation: one of them is statement while the other one is prepared statement. Solution in final project: class no longer been used. New class will replace the old one.
14,15,16,17,18	1.a(ii)	Same as 5. All these functions are reusing other's code, thus can be refactored in the same way: abstract the clone part and make it a new function. However, these 2 classes will no longer be used in final project; New classes will replace them.
19	4.b(i)	Accidently clone of absolutely same code. Should make it an abstract class for both SearchDishes and SearchRestaurant.
20	1.a(i)	Reusing existing code by copying and pasting with different SQL Query only. Solution in final project: abstract the clone part as a new function.
21	1.a(ii)	Reuse similar solutions: used in different situations: one of them sort integer array while the other one sort Long array. Solution in final project: class no longer been used. Use "compare to" function for now.
22	1.a(ii)	Those printing functions can check whether the output fits the expectation. However, similar solutions have been used. Solution in final project: Use Junit instead.
All the clones from 23 – 39 happens in 4 classes that three of them are merged and created the last class while the three resource classes are not deleted. What's more, the function inside those four classes are duplicated as well. Thus, three resource classes will be deleted while the merged class will be refactored.		
23 - 25	1.a(i)	Reusing existing code by copying and pasting with object been modified. Solution in final project: abstract the clone part as a new function.
26	1.a(ii)	Reuse similar code as solutions of similar problems. Solution in final project: abstract the clone part as a new function.
27, 28	1.a(i)	Same as 23-25
29	1.a(ii)	Same as 26
30	1.a(i)	Same as 23-25
31	1.a(ii)	Same as 26
32	1.a(ii)	Same as 26
33	4.b(i) (Kind of)	Did not delete the resource class after merging
34, 35	1.a(i)	Same as 23-25
36	1.a(ii)	Those printing functions can check whether the output fits the expectation. However, similar solutions have been

		used. Solution in final project: Use Junit instead.
37 - 39	1.a(i)	Same as 23-25
40	1.a(i)	Reuse of same solution with minor modification. Solution in final project: abstract the clone part as a new function.
41	2.a(i)	Use similar way to do the test in similar classes can avoid the risk of going wrong. Solution in final project: Use Junit instead.
42 - 48	2.a(i)	Use similar way to do the test in similar classes can avoid the risk of going wrong. All of these can be modified together in the same way. Solution in final project: Use Junit instead.

4.2 Refactoring (5 marks)

ReadMe for 4.2:

Lots of classes have been changed after milestone 5.
New classes created after milestone 5 but didn't refactor are in folder "old"
New classes refactored based on the classes in folder "old" are in folder "new"

4.2 Refactoring:

There are several refactors in the project. Code before refactoring are in folder "old" while code after refactoring are in folder "new" or can be found in the project.

Proofs are given below:

Refactoring number 1, 2:

Before: package "google" class "getDistance", Line 38 & 42. Could be found in milestone 5.

After: package "google" class "getDistance", Line 39, 55, 58, 88, 89

Type of refactoring: Replace magic number with Symbolic Constant
Replace array with object

Prototype before refactoring:

```
26      /**
27       * When Call this function, you will get the String of distance between origin and destination
28       * @param origin
29       * @param destination
30       * @return a string
31       */
32      public String getDistanceString(String origin, String destination) {
33
34          if (context != null) {
35              System.out.println("Connect");
36              String [] orig = {origin};
37              String [] dest = {destination};
38              DirectionsResult result = DirectionsApi.getDirections(context, orig[0], dest[0]).awaitIgnoreError();
39              Distance distance = new Distance();
40              distance.getClass().getResource(result.routes.clone().toString());
41              DistanceMatrix distanceMatrix = DistanceMatrixApi.getDistanceMatrix(context, orig, dest).awaitIgnoreError();
42              getDistance = distanceMatrix.rows[0].elements[0].distance.humanReadable;
43              //distanceFloat = Float.valueOf(distanceMatrix.rows[0].elements[0].distance.humanReadable.toString());
44
45          }
46          return getDistance;
47
48      }
```

Magic number “0” is used in line 38 & 42.
Array “orig[0], dest[0], distanceMatrix.rows[0].element[0]” are used in line 38 & 42.

Prototype after refactoring:

```
36          /**
37           *  The first element of a list.
38           */
39          private final int FIRST_ELEMENT = 0;
40
```

```
83      /**
84       * Helper function that generate the first element from a String list.
85       *
86       * @param input the input String list
87       * @return
88       */
89  @   private String getFirstElementOfStringList(final String[] input) { return input[FIRST_ELEMENT]; }
92
93      /**
94       * Helper function that generate the first element from a DistanceMatrix.
95       *
96       * @param input the input DistanceMatrix
97       * @return
98       */
99  @   private DistanceMatrixElement getElement(final DistanceMatrix input) {
100             return input.rows[FIRST_ELEMENT].elements[FIRST_ELEMENT];
101         }
102
```

```
41      /**
42       * When Call this function,
43       * you will get the String of distance between origin and destination.
44       * @ParameterNames origin
45       * @ParameterNames destination
46       * @return the string that contains the distance between two inputs
47       */
48      public String getStringDistance(final String origin, final String destination) {
49
50          if (context != null) {
51              Distance distance = new Distance();
52              String[] orig = {origin};
53              String[] dest = {destination};
54              DirectionsResult result = DirectionsApi.getDirections(
55                      context, getFirstElementOfStringList(orig), getFirstElementOfStringList(dest)).awaitIgnoreError();
56              distance.getClass().getResource(Arrays.toString(result.routes.clone()));
57              DistanceMatrix distanceMatrix = DistanceMatrixApi.getDistanceMatrix(context, orig, dest).awaitIgnoreError();
58              getDistance = getElement(distanceMatrix).distance.humanReadable;
59          }
60          return getDistance;
61
62      }
63
```

Magic number “0” is replaced by FIRST_ELEMENT in line 39.
Array “orig[0], dest[0], distanceMatrix.rows[0].element[0]” are replaced by functions that will return the array “getElement” & “getFirstElementOfStringList” in line 89 & 99, and is called by the getStringDistance function in line 55 & 58.

Reason for refactoring: existence of magic number & array directly calls
How this improve the code: magic number & array directly calls are eliminated; better code smells.

Refactoring number 3, 4:

Before: package “Payment” class “CreditCardVerification”, Line 89-108 & 156-176. Could be found in milestone 5.

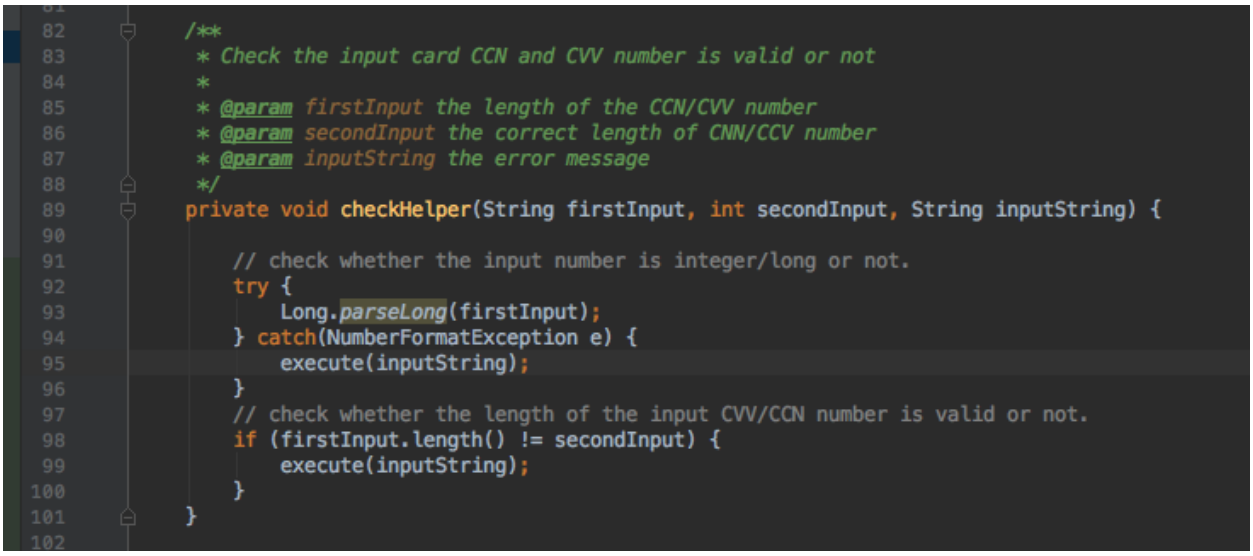
After: package “Payment” class “CreditCardVerification”, Line 89-100

Type of refactoring: Extract method
Consolidate Conditional Expression

Prototype before refactoring:

```
84      /**
85       * Checks the validity of the entered Credit Card number using the length of numbers entered
86       * @param C Credit Card details entered by user
87       * @return true if the Credit Card number entered is exactly 16 digits
88       */
89      public boolean isCCNumberLengthValid(CreditCardVerification C){
90
91          try {
92
93              if (C.ccNumber.length() < stdCCNumberDigit)
94                  throw new RuntimeException();
95
96
97              if (C.ccNumber.length() > stdCCNumberDigit)
98                  throw new RuntimeException();
99          }
100
101         catch(RuntimeException e) {
102
103             message = "Error: CREDITCARD NUMBER ENTERED IS INVALID";
104         }
105
106         return  true;
107
108     }
150
151     /**
152      * Checks the validity of the entered CVV number using the length of numbers entered
153      * @param C Credit Card details entered by user
154      * @return true is the creditCardVerification card verification number entered is exactly 3 digits
155      */
156     public boolean isCVVLengthValid(CreditCardVerification C){
157
158         try {
159
160             if (C.CVV.length() < stdCVVDigits)
161                 throw new RuntimeException();
162
163
164             if (C.CVV.length() > stdCVVDigits)
165                 throw new RuntimeException();
166         }
167
168         catch(RuntimeException e) {
169
170             message = "Error: CVV ENTERED IS INVALID";
171
172         }
173
174         return  true;
175
176     }
```

Line 93-99 & 160-166: Sequence of conditional tests that have same result;
Class ‘isCCNumberLengthValid’ & ‘isCVVLengthValid’ are duplicated.



Prototype after refactoring:

```
82      /**
83       * Check the input card CCN and CVV number is valid or not
84       *
85       * @param firstInput the length of the CCN/CVV number
86       * @param secondInput the correct length of CNN/CCV number
87       * @param inputString the error message
88       */
89      private void checkHelper(String firstInput, int secondInput, String inputString) {
90
91          // check whether the input number is integer/long or not.
92          try {
93              Long.parseLong(firstInput);
94          } catch(NumberFormatException e) {
95              execute(inputString);
96          }
97          // check whether the length of the input CVV/CCN number is valid or not.
98          if (firstInput.length() != secondInput) {
99              execute(inputString);
100         }
101     }
102
```

Sequence of conditional tests that have same result are replaced by the test at Line 98-100.
Class ‘isCCNumberLengthValid’ & ‘isCVVLengthValid’ are abstracted and formed the new method ‘checkHelper’

Reason for refactoring: existence of duplicated code & tests that have same result

How this improve the code: duplicated code & tests that have same result are extracted and thus eliminated; better code smells.

Refactoring number 5, 6:

Before: package “database” class “Adduser”, Line 46-76. Could be found in milestone 5.

After: package “database” class “Adduser”, Line 56, 65-77

Type of refactoring: Extract method
Consolidate Duplicate Conditional Fragments

Prototype before refactoring:

```
46          try {
47              // check if the username has been used or not
48              String checkQuery = "select username from users where username = ?";
49              PreparedStatement checkStmt = connect.coon.prepareStatement(checkQuery);
50              checkStmt.setString( parameterIndex: 1,username);
51              ResultSet rs = checkStmt.executeQuery();
52              if (rs.next())
53              {
54                  message = "This username has been used";
55                  System.out.println(message);
56                  connect.coon.close();
57              }
58
59
60              else
61              {
62                  // if not used username, then insert them to database
63                  String insertQuery = "insert into users (username, passwords, user_status) values(?,?,?)";
64                  un = username;
65                  st = status;
66                  PreparedStatement ppStmt = connect.coon.prepareStatement(insertQuery);
67                  ppStmt.setString( parameterIndex: 1, username);
68                  ppStmt.setString( parameterIndex: 2, userPassword);
69                  ppStmt.setString( parameterIndex: 3, status);
70                  int affected = ppStmt.executeUpdate();
71                  if (affected > 0) {
72                      message = "You have successfully signed up";
73                      System.out.println(message);
74                      connect.coon.close();
75                  }
76              }
77          }catch (SQLException e) {
```

Line 56 & 74: code is in all branches of a conditional expression
Line 48-51 & 66-69: duplicated code
Line 54-55 & 72-73: duplicated code

Prototype after refactoring:

```
64
65      private void insideHelper(String input, String username){
66          try {
67              statement = connect.coon.prepareStatement(input);
68              statement.setString( parameterIndex: 1, username);
69          } catch (SQLException e) {
70              e.printStackTrace();
71          }
72      }
73
74      private void setMessage(String input){
75          message = input;
76          System.out.println(message);
77      }
```

```
31      public void addUser(String username, String userPassword, String status)
32      {
33          String Query;
34          connect.connect();
35          if(connect.coon != null) {
36              try {
37                  // check if the username has been used or not
38                  Query = "select username from users where username = ?";
39                  this.insideHelper(Query,username);
40                  ResultSet rs = statement.executeQuery();
41                  if (rs.next()) {
42                      this.setMessage("This username has been used");
43                  } else {
44                      // if not used username, then insert them to database
45                      Query = "insert into users (username, passwords, user_status) values(?,?,?)";
46                      un = username;
47                      st = status;
48                      this.insideHelper(Query,username);
49                      statement.setString( parameterIndex: 2, userPassword);
50                      statement.setString( parameterIndex: 3, status);
51                      int affected = statement.executeUpdate();
52                      if (affected > 0) {
53                          this.setMessage("You have successfully signed up");
54                      }
55                  }
56                  connect.coon.close();
57              } catch (SQLException e) {
58                  e.printStackTrace();
59              }
```

Duplicated codes are now abstracted and new methods “insideHelper” and “setMessage” are created.
Line 56: Code appeared in all branches are now moved outside of the expression.

Reason for refactoring: existence of duplicated code & code appeared in all branches

How this improve the code: duplicated codes are extracted & code appeared in all branches are moved outside of the expression and thus eliminated; better code smells

Refactoring number 7:

Before: package “database” class “RestaurantFunction”, Line 36-44 & 80-88

After: package “database” class “Restaurants”, Line 86-104

Type of refactoring: Extract method

Prototype before refactoring:

```
26      public void addRestaurant(Restaurant restaurant)
27      {
28          GoConnection connection = new GoConnection();
29          connection.connect();
30          if(connection.coon!=null)
31          {
32              try {
33                  String addQuery = "insert into restaurants (user_id, restaurant_name, license_id, open_time, close_time,phone_num, e_mail_ad
34                  PreparedStatement ppStmt = connection.coon.prepareStatement(addQuery);
35
36                  ppStmt.setInt( parameterIndex: 1,restaurant.getUser_id());
37                  ppStmt.setString( parameterIndex: 2,restaurant.getRestaurant_name());
38                  ppStmt.setString( parameterIndex: 3,restaurant.getLicense_id());
39                  ppStmt.setTime( parameterIndex: 4,restaurant.getOpen_time());
40                  ppStmt.setTime( parameterIndex: 5,restaurant.getClose_time());
41                  ppStmt.setString( parameterIndex: 6,restaurant.getPhone_num());
42                  ppStmt.setString( parameterIndex: 7,restaurant.getE_mail_address());
43                  ppStmt.setTime( parameterIndex: 8, restaurant.getWaiting_time());
44                  ppStmt.setFloat( parameterIndex: 9,restaurant.getAvg_price());
45
46                  int value = ppStmt.executeUpdate();
47                  if(value>0)
48                  {
49                      message = "The restaurant information has been saved";
50                      connection.coon.close();
51                  }
52              }catch (SQLException e)
53              {
54                  message = e.fillInStackTrace().toString();
55              }
56          }
57          else
58          {
59              message = "lost connection";
60          }
61      }
```

```
63      public void updateRestaurant(int user_id, Restaurant newRestaurant)
64      {
65          GoConnection connection = new GoConnection();
66          connection.connect();
67          if(connection.coon!=null)
68          {
69              try{
70                  String query = "UPDATE restaurants\n" +
71                          "restaurant_name = ?\n" +
72                          "license_id = ?\n" +
73                          "open_time = ?\n" +
74                          "close_time = ?\n" +
75                          "phone_num = ?\n" +
76                          "e_mail_address=?\n" +
77                          "waiting_time = ?\n" +
78                          "avg_price = ?\n" +
79                          "WHERE user_id = ?";
80                  PreparedStatement preparedStatement = connection.coon.prepareStatement(query);
81                  preparedStatement.setString( parameterIndex: 1,newRestaurant.getRestaurant_name());
82                  preparedStatement.setString( parameterIndex: 2,newRestaurant.getLicense_id());
83                  preparedStatement.setTime( parameterIndex: 3, newRestaurant.getOpen_time());
84                  preparedStatement.setTime( parameterIndex: 4,newRestaurant.getClose_time());
85                  preparedStatement.setString( parameterIndex: 5,newRestaurant.getE_mail_address());
86                  preparedStatement.setTime( parameterIndex: 6, newRestaurant.getWaiting_time());
87                  preparedStatement.setFloat( parameterIndex: 7,newRestaurant.getAvg_price());
88                  preparedStatement.setInt( parameterIndex: 8,user_id);
89
90                  int value = preparedStatement.executeUpdate();
91                  if(value>0)
92                  {
93                      message = "the restaurant information has been update";
```

Line 36-44 & 80-88: duplicated code

Prototype after refactoring:

```
69      /**
70       * Helper function that set the input preparation
71       *
72       * @param prep the input preparation
73       * @param restaurant the resource
74       * @param which detect whether it is add or update
75       * @param first the first question mark
76       * @param second the second question mark
77       * @param third the third question mark
78       * @param forth the forth question mark
79       * @param fifth the fifth question mark
80       * @param sixth the sixth question mark
81       * @param seventh the seventh question mark
82       * @param eighth the eighth question mark
83       * @param ninth the ninth question mark
84       * @param user_id if it is update then we need an user_id
85       */
86      private void setPrep(PreparedStatement prep, Restaurant restaurant,int which, int first, int second, int third, int forth, int fifth,
87                           int sixth, int seventh, int eighth, int ninth, int user_id) {
88          try {
89              if (which == 1) {
90                  prep.setInt(first, restaurant.getUser_id());
91              } else {
92                  prep.setInt(first, user_id);
93              }
94              prep.setString(second, restaurant.getRestaurant_name());
95              prep.setString(third, restaurant.getLicense_id());
96              prep.setTime(forth, restaurant.getOpen_time());
97              prep.setTime(fifth, restaurant.getClose_time());
98              prep.setString(sixth, restaurant.getPhone_num());
99              prep.setString(seventh, restaurant.getE_mail_address());
100             prep.setTime(eighth, restaurant.getWaiting_time());
101             prep.setFloat(ninth, restaurant.getAvg_price());
102         } catch (SQLException e) {
103             message = e.fillInStackTrace().toString();
104         }
105     }
```

Line 86-104: Duplicated code are abstracted and formed a new method “setPrep”.

Reason for refactoring: existence of duplicated code

How this improve the code: duplicated codes are extracted and thus eliminated; better code smells.

Refactoring number 8:

Before: package “database” class “RestaurantFunction”, Line 157-167 & 120-132

After: package “database” class “Restaurants”, Line 158-188

Type of refactoring: Extract method

Prototype before refactoring:

```
144     public Restaurant getRestaurant(int user_id)
145     {
146         GoConnection connection = new GoConnection();
147         connection.connect();
148         if(connection.coon!=null)
149         {
150             try{
151                 String query = "select restaurant_name, license_id, open_time, close_time, phone_num, e_mail_address, waiting_time, avg_price \
152                         "from restaurants WHERE user_id = "+user_id;
153                 Statement statement = connection.coon.createStatement();
154                 ResultSet resultSet = statement.executeQuery(query);
155                 if(resultSet.next())
156                 {
157                     restaurant_name = resultSet.getString( columnIndex: 1);
158                     license_id = resultSet.getString( columnIndex: 2);
159                     open_time = resultSet.getTime( columnIndex: 3);
160                     close_time = resultSet.getTime( columnIndex: 4);
161                     phone_num = resultSet.getString( columnIndex: 5);
162                     e_mail_address = resultSet.getString( columnIndex: 6);
163                     waiting_time = resultSet.getTime( columnIndex: 7);
164                     avg_price = resultSet.getFloat( columnIndex: 8);
165                     restaurant = new Restaurant(user_id, restaurant_name,license_id,open_time,close_time,phone_num,e_mail_address);
166                     restaurant.setWaiting_time(waiting_time);
167                     restaurant.setAvg_price(avg_price);
168                 }
169                 connection.coon.close();
170 
171             }catch (SQLException e)
172             {
173                 message = e.fillInStackTrace().toString();
```

```
107     public ArrayList<Restaurant> displayAllRestaurants() {
108         GoConnection connection = new GoConnection();
109         connection.connect();
110         displayRestaurants = new ArrayList<>();
111         if(connection.coon!=null)
112         {
113             try{
114                 String query = "select user_id, restaurant_name, license_id, open_time, close_time, phone_num, e_mail_address, waiting_time, a
115                         "from restaurants";
116                 Statement statement = connection.coon.createStatement();
117                 ResultSet resultSet = statement.executeQuery(query);
118                 while(resultSet.next())
119                 {
120                     user_id = resultSet.getInt( columnIndex: 1);
121                     restaurant_name = resultSet.getString( columnIndex: 2);
122                     license_id = resultSet.getString( columnIndex: 3);
123                     open_time = resultSet.getTime( columnIndex: 4);
124                     close_time = resultSet.getTime( columnIndex: 5);
125                     phone_num = resultSet.getString( columnIndex: 6);
126                     e_mail_address = resultSet.getString( columnIndex: 7);
127                     waiting_time = resultSet.getTime( columnIndex: 8);
128                     avg_price = resultSet.getFloat( columnIndex: 9);
129                     Restaurant temp = new Restaurant(user_id,restaurant_name,license_id,open_time,close_time,phone_num,e_mail_address);
130                     temp.setWaiting_time(waiting_time);
131                     temp.setAvg_price(avg_price);
132                     displayRestaurants.add(temp);
133                 }
134                 connection.coon.close();
135 
136             }catch (SQLException e)
137             {
```

Line 157-167 & 120-132: duplicated code

Prototype after refactoring:

```
158     private void setRestaurantHelper(ResultSet resultSet, int which, int first, int second, int third, int forth, int fifth,
159                                      int sixth, int seventh, int eighth, int ninth, int user_id) {
160         try {
161             if (which == 1) {
162                 user_id = resultSet.getInt(first);
163             }
164
165             restaurant_name = resultSet.getString(second);
166             license_id = resultSet.getString(third);
167             open_time = resultSet.getTime(forth);
168             close_time = resultSet.getTime(fifth);
169             phone_num = resultSet.getString(sixth);
170             e_mail_address = resultSet.getString(seventh);
171             waiting_time = resultSet.getTime(eighth);
172             avg_price = resultSet.getFloat(ninth);
173
174             Restaurant temp = new Restaurant(user_id,restaurant_name,license_id,open_time,close_time,phone_num,e_mail_address);
175             temp.setWaiting_time(waiting_time);
176             temp.setAvg_price(avg_price);
177
178             if (which == 1) {
179                 displayRestaurants.add(temp);
180             } else {
181                 restaurant = temp;
182             }
183
184         } catch (SQLException e) {
185             message = e.fillInStackTrace().toString();
186         }
187     }
```

Line 158-188: Duplicated code are abstracted and formed a new method “setRestaurantHelper”.

Reason for refactoring: existence of duplicated code

How this improve the code: duplicated codes are extracted and thus eliminated; better code smells.

Refactoring number 9:

Before: package “database” class “MenuFunction”, Line 33-37 & 38-41 & 56-59 & 90-93 & 121-125

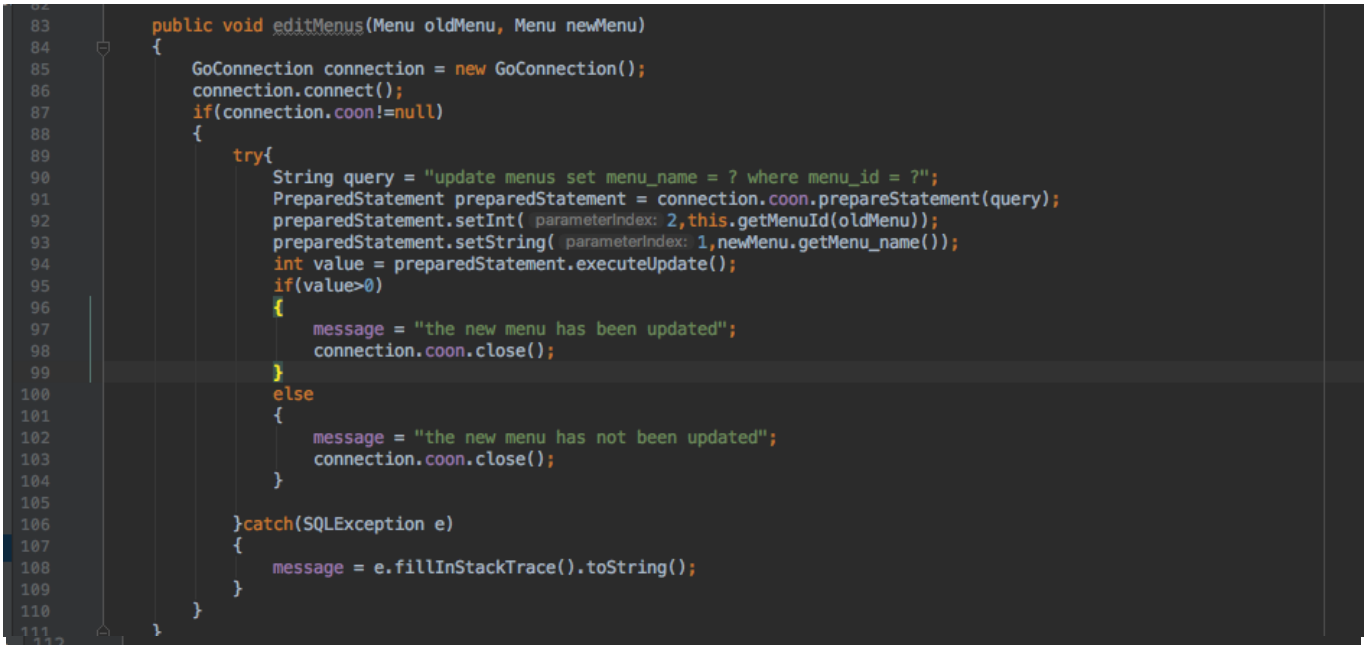
After: package “database” class “Menus”, Line 70-97

Type of refactoring: Extract method

Prototype before refactoring:

```
17
18      public void addMenu(Menu menu)
19      {
20          GoConnection connection = new GoConnection();
21          if(menu.getMenu_name() == null)
22          {
23              message = "menuName can not be null, please check";
24              return;
25          }
26          else
27          {
28              connection.connect();
29              try
30              {
31                  if(connection.coon!=null)
32                  {
33                      String checkQuery = "select menu_name from menus where restaurant_id = ? and menu_name = ?";
34                      PreparedStatement checkppStmt = connection.coon.prepareStatement(checkQuery);
35                      checkppStmt.setInt( parameterIndex: 1,menu.getRestaurants_id());
36                      checkppStmt.setString( parameterIndex: 2,menu.getMenu_name());
37                      ResultSet checkResult = checkppStmt.executeQuery();
38                      String checkQuery1 = "select id from restaurants where id = ?";
39                      PreparedStatement checkppStmt1 = connection.coon.prepareStatement(checkQuery1);
40                      checkppStmt1.setInt( parameterIndex: 1,menu.getRestaurants_id());
41                      ResultSet checkResult1 = checkppStmt1.executeQuery();
```

```
42                      if(checkResult.next())
43                      {
44                          message = "This menu name has already used";
45                          connection.coon.close();
46                      }
47                      else if(!checkResult1.next())
48                      {
49                          message = "Did not find the restaurant id";
50                          connection.coon.close();
51                      }
52                      else
53                      {
54                          System.out.println(menu);
55
56                          String addMenuQuery = "INSERT INTO menus (restaurant_id, menu_name) VALUES (?,?)";
57                          PreparedStatement ppStmt = connection.coon.prepareStatement(addMenuQuery);
58                          ppStmt.setInt( parameterIndex: 1, menu.getRestaurants_id());
59                          ppStmt.setString( parameterIndex: 2, menu.getMenu_name());
60                          int affected = ppStmt.executeUpdate();
61                          if (affected > 0)
62                          {
63                              message = "You have successfully add the menu "+menu.getMenu_name();
64                              System.out.println(message);
65                              connection.coon.close();
66                          }
67                      }
68
69                  }
70              }
71              catch (SQLException e)
72              {
```



```
83      public void editMenus(Menu oldMenu, Menu newMenu)
84      {
85          GoConnection connection = new GoConnection();
86          connection.connect();
87          if(connection.coon!=null)
88          {
89              try{
90                  String query = "update menus set menu_name = ? where menu_id = ?";
91                  PreparedStatement preparedStatement = connection.coon.prepareStatement(query);
92                  preparedStatement.setInt( parameterIndex: 2,this.getMenuId(oldMenu));
93                  preparedStatement.setString( parameterIndex: 1,newMenu.getMenu_name());
94                  int value = preparedStatement.executeUpdate();
95                  if(value>0)
96                  {
97                      message = "the new menu has been updated";
98                      connection.coon.close();
99                  }
100                 else
101                 {
102                     message = "the new menu has not been updated";
103                     connection.coon.close();
104                 }
105
106             }catch(SQLException e)
107             {
108                 message = e.fillInStackTrace().toString();
109             }
110         }
111     }
```

```
112
113     public int getMenuId(Menu menu)
114     {
115         GoConnection connection = new GoConnection();
116         connection.connect();
117         int menu_id = 0;
118         if(connection.coon!=null)
119         {
120             try{
121                 String query = "select menu_id from menus where restaurant_id = ? and menu_name = ?";
122                 PreparedStatement statement = connection.coon.prepareStatement(query);
123                 statement.setInt( parameterIndex: 1,menu.getRestaurants_id());
124                 statement.setString( parameterIndex: 2,menu.getMenu_name());
125                 ResultSet resultSet = statement.executeQuery();
126                 if(resultSet.next())
127                 {
128                     menu_id = resultSet.getInt( columnIndex: 1);
129                 }
130                 else
131                 {
132                     message = "The menu id not found";
133                 }
134                 connection.coon.close();
135             }catch (SQLException e)
136             {
137                 message = e.fillInStackTrace().toString();
138             }
139         }
140         else
141         {
142             message = "lost connection";
```

Line 33-37 & 38-41 & 56-59 & 90-93 & 121-125: duplicated code

Prototype after refactoring:

```
70  public PreparedStatement setStatement(String inputQuery, int id, String name, boolean which, int setInt, int setString){
71      try {
72          PreparedStatement statement = connection.coon.prepareStatement(inputQuery);
73          statement.setInt(setInt, id);
74          if(which) {
75              statement.setString(setString, name);
76          }
77          return statement;
78      } catch (SQLException e) {
79          e.printStackTrace();
80      }
81      return null;
82  }
83
84  /**
85   * Gathering the Result Set from the input PreparedStatement
86   * @param inputStatement the input PreparedStatement
87   * @return the Result Set
88   */
89  public ResultSet gatherResultSet(PreparedStatement inputStatement){
90      ResultSet resultSet = null;
91      try {
92          resultSet = inputStatement.executeQuery();
93      } catch (SQLException e) {
94          e.printStackTrace();
95      }
96      return resultSet;
97  }
```

Line 70-97: Duplicated code are abstracted and formed two new methods “setStatement” & “gatherResultSet”.

Reason for refactoring: existence of duplicated code

How this improve the code: duplicated codes are extracted and thus eliminated; better code smells.

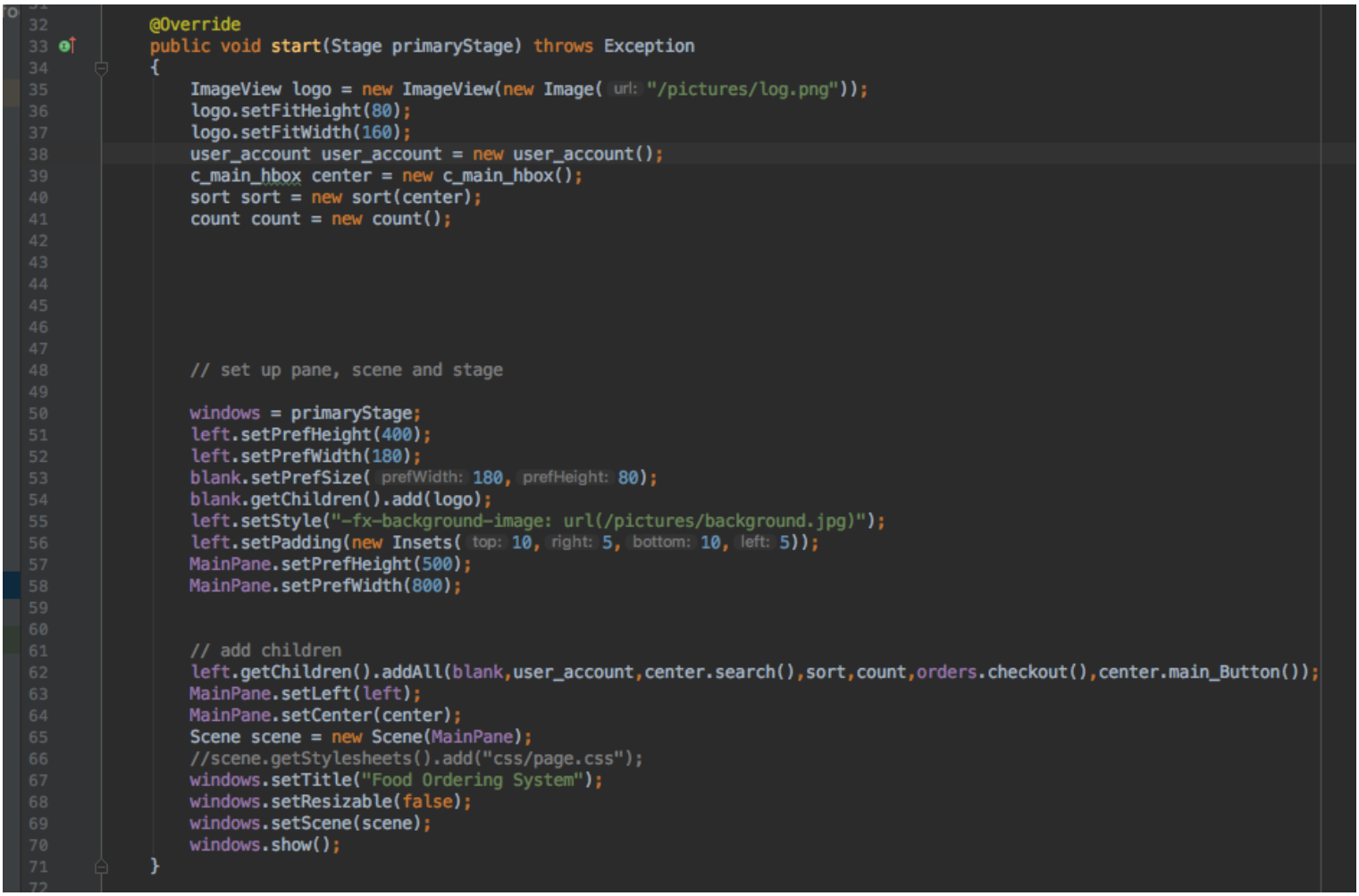
Refactoring number 10:

Before: package “UI” class “main_frame”, Line 50-70. Could be found in milestone 5.

After: package “UI” class “main_frame”, Line 70-97

Type of refactoring: Extract method

Prototype before refactoring:



```
32      @Override
33      public void start(Stage primaryStage) throws Exception
34      {
35          ImageView logo = new ImageView(new Image( url: "/pictures/log.png"));
36          logo.setFitHeight(80);
37          logo.setFitWidth(160);
38          user_account user_account = new user_account();
39          c_main_hbox center = new c_main_hbox();
40          sort sort = new sort(center);
41          count count = new count();
42
43
44
45
46
47
48          // set up pane, scene and stage
49
50          windows = primaryStage;
51          left.setPrefHeight(400);
52          left.setPrefWidth(180);
53          blank.setPrefSize( prefWidth: 180, prefHeight: 80);
54          blank.getChildren().add(logo);
55          left.setStyle("-fx-background-image: url(/pictures/background.jpg)");
56          left.setPadding(new Insets( top: 10, right: 5, bottom: 10, left: 5));
57          MainPane.setPrefHeight(500);
58          MainPane.setPrefWidth(800);
59
60
61          // add children
62          left.getChildren().addAll(blank,user_account,center.search(),sort,count,orders.checkout(),center.main_Button());
63          MainPane.setLeft(left);
64          MainPane.setCenter(center);
65          Scene scene = new Scene(MainPane);
66          //scene.getStylesheets().add("css/page.css");
67          windows.setTitle("Food Ordering System");
68          windows.setResizable(false);
69          windows.setScene(scene);
70          windows.show();
71      }
72
```

Code fragment that can be group together:

Line 36-37
Line 51, 52, 55, 56, 62
Line 53, 54
Line 57, 58, 63, 64
Line 67 - 70

Prototype after refactoring:

```
96      /** Set up the FOS Logo */
97
98      private void setLogo() {
99          logo.setFitHeight(80);
100         logo.setFitWidth(160);
101     }
102
103     /** Set up the blank for the FOS logo */
104     private void setBlank() {
105         blank.setPrefSize( prefWidth: 180,  prefHeight: 80);
106         blank.getChildren().add(logo);
107     }
108
109     /** Set up the Left area */
110     private void setLeft() {
111         left.setPrefHeight(400);
112         left.setPrefWidth(180);
113         left.setStyle("-fx-background-image: url(/pictures/background.jpg)");
114         left.setPadding(new Insets( top: 10,  right: 5,  bottom: 10,  left: 5));
115         // add children
116         left.getChildren().addAll(blank, user_account, center.search(),
117                 center.setsort(), count, orders.checkout(), center.main_Button());
118     }
119
120     /** Set up the Main BorderPane */
121     private void setMainPane() {
122         MainPane.setPrefHeight(500);
123         MainPane.setPrefWidth(800);
124         MainPane.setLeft(left);
125         MainPane.setCenter(center);
126     }
127
128     /**
129      * Set up the stage
130      */
131     private void setWindows() {
132         windows.setTitle("Food Ordering System");
133         windows.setResizable(false);
134         windows.setScene(scene);
135         windows.show();
136     }
```

Line 36-37 is turned into a method called “setLogo”
Line 51, 52, 55, 56, 62 is turned into a method called “setLeft”
Line 53, 54 is turned into a method called “setBlank”
Line 57, 58, 63, 64 is turned into a method called “setMainPane”
Line67 – 70 is turned into a method called “setWindows”

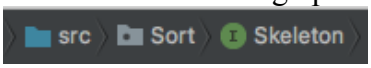
Reason for refactoring: different setting functions are merged in one function

How this improve the code: code fragments with same function are turned into new methods; better code smells.

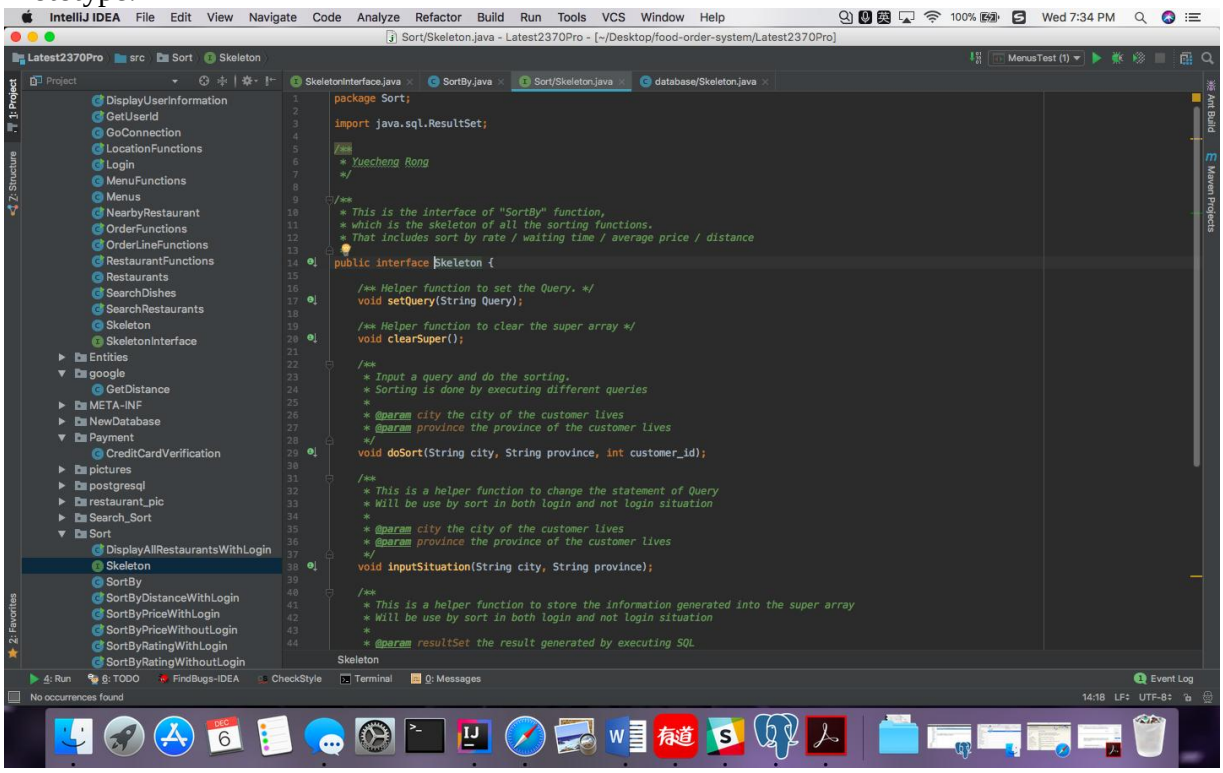
4.3 Gang of Four Design Patterns (4 marks)

The Gang of Four pattern we used: **Command Pattern**

1) The first command design pattern implemented in the project is "**Skeleton**"

Location: 

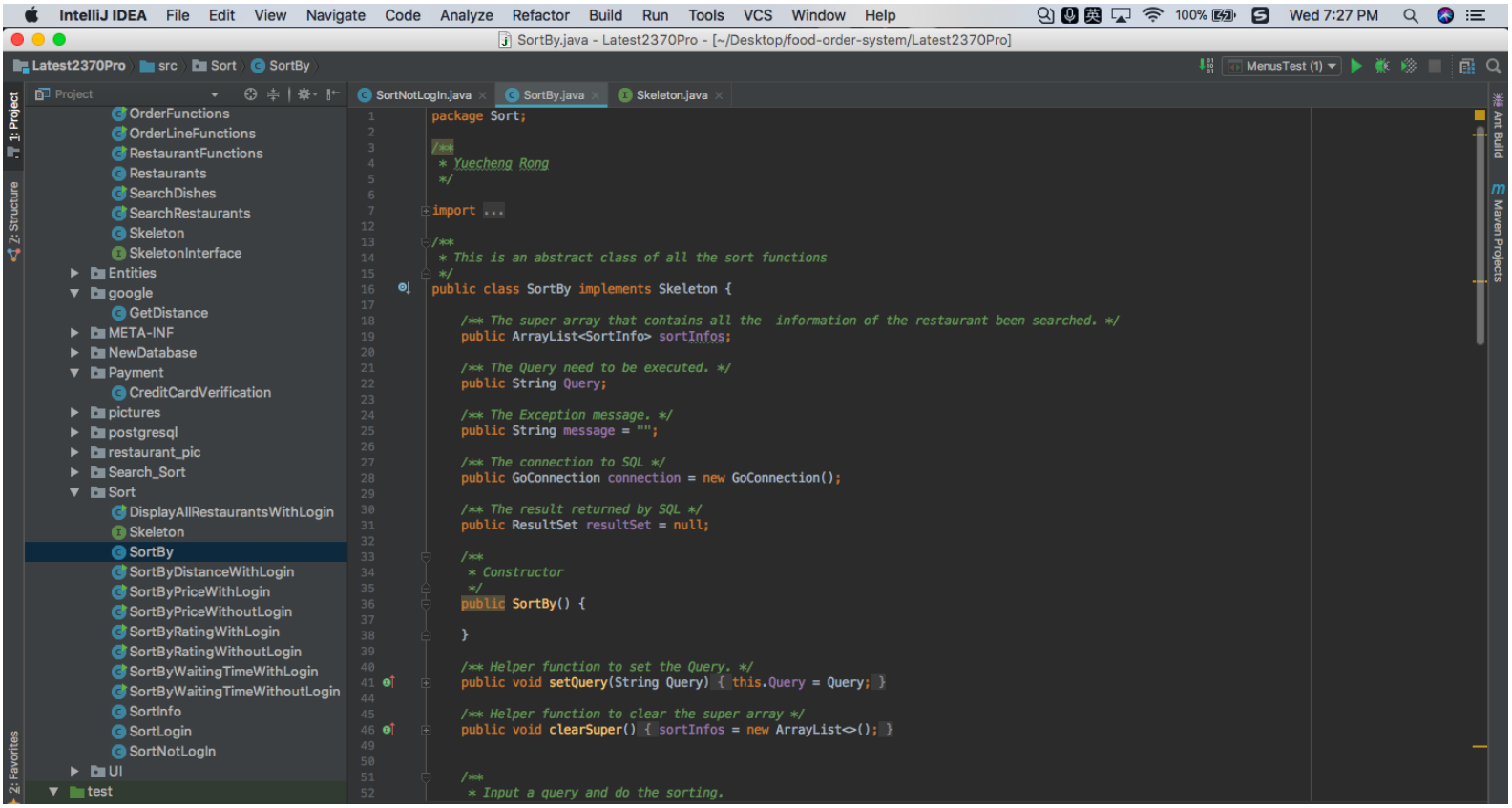
Prototype:



The class implements the interface: "**SortBy**", which is the abstract class for "SortLogIn" and "SortNotLogin"

Location: src > Sort > SortBy

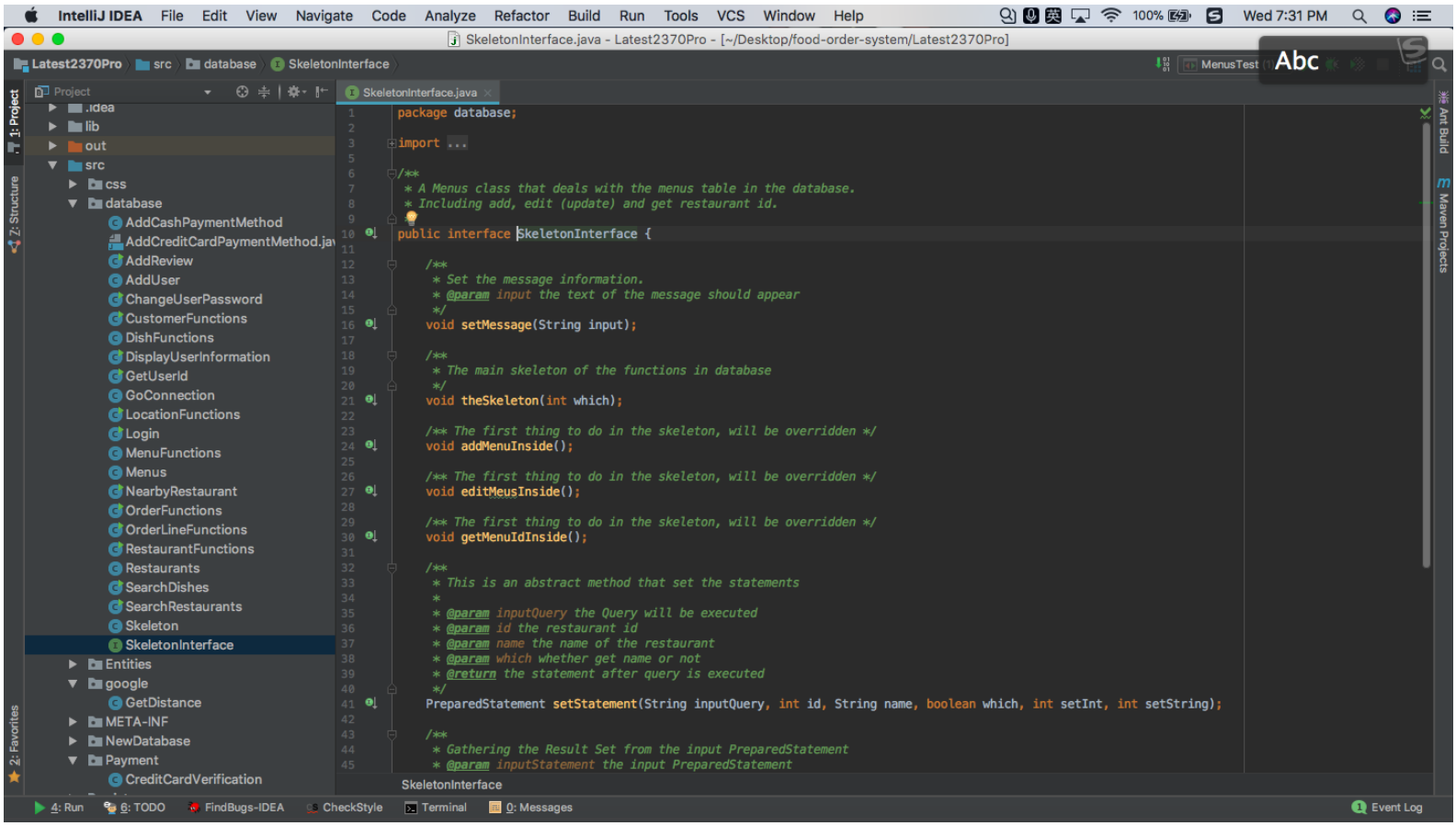
Prototype:



2) The second command design pattern implemented in the project is "**SkeletonInterface**"

Location: src > database > SkeletonInterface

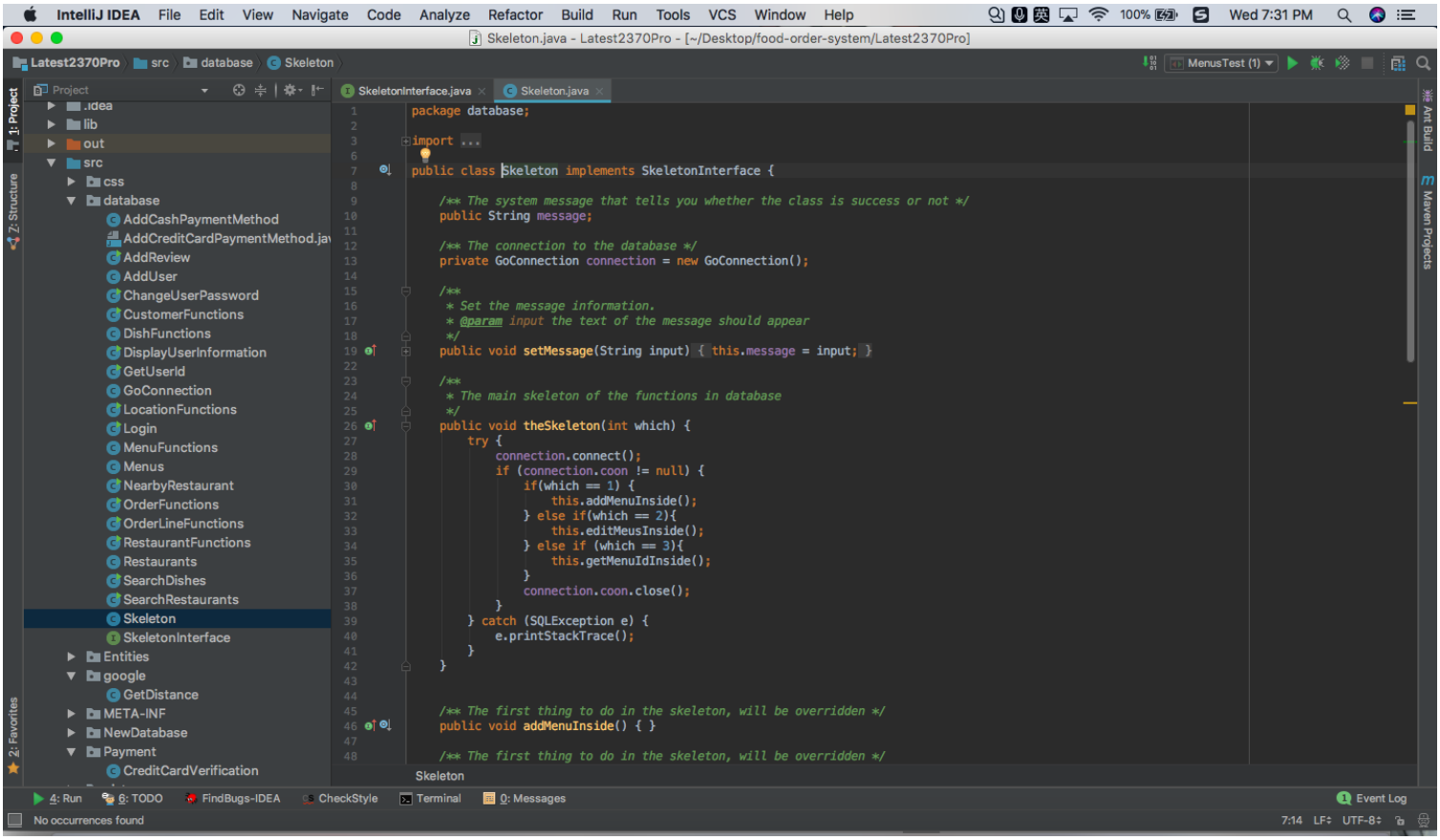
Prototype:



The class implements the interface: "**Skeleton**", which is the abstract for class "Menus"

Location: src > Sort > SortBy

Prototype:



5. Complete Implementation and Product Delivery (33 marks)

5.1 Naming Conventions (1 mark)

Proper naming conventions were followed in our programming. In particular, these techniques were employed:

- Naming variables with camel case or using underscores
- Naming variables with a descriptive names

5.2 Commenting (1 mark)

Proper commenting conventions were followed in our programming. In particular, these techniques were employed

- Commenting on confusing code
- Commenting about date, time, and who had wrote and edited the code

5.3 Pretty-printing of the source (2 marks)

We used the pretty_printing by using Astyle:

1. We firstly download the Compressed Astyle file from website.
2. We installed it step by step from online document and run it by terminal.
3. We run the command "astyle --style=allman /*.java" recursively in our folder, then get the good printing
4. We deleted the duplicated file (old file), then the pretty-printing process was done.

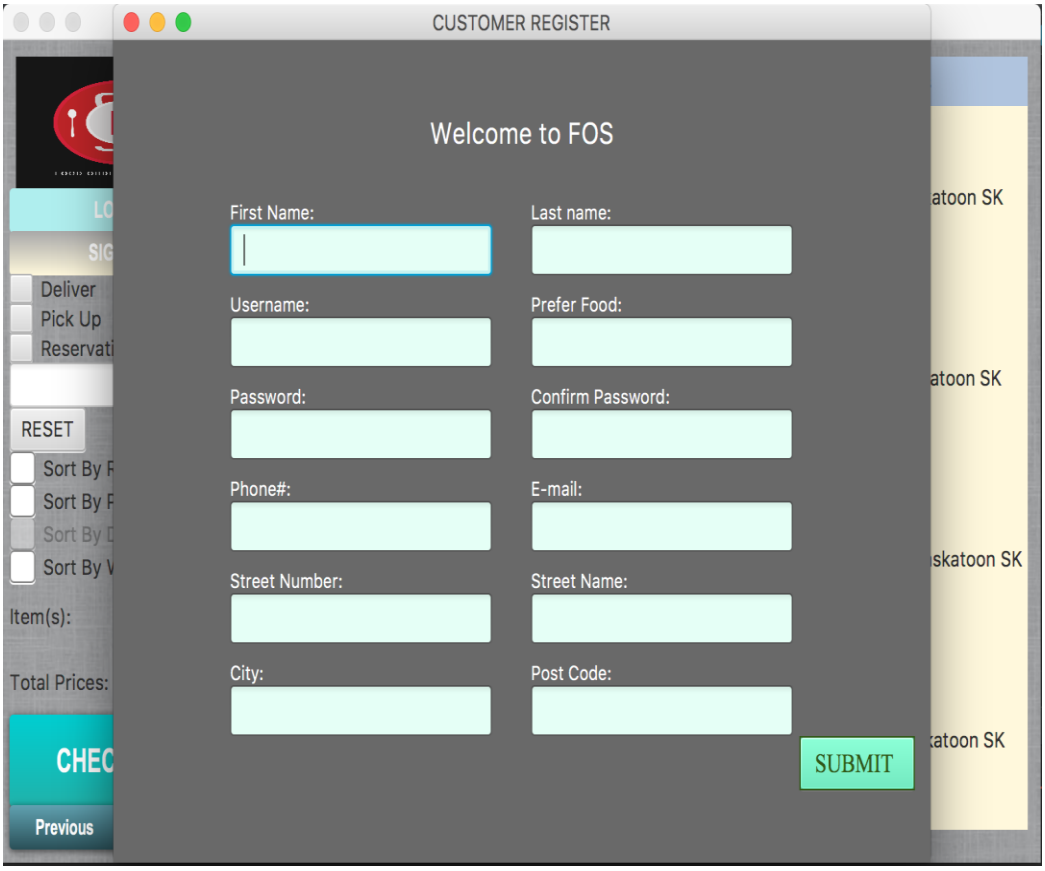
5.4 Usability Engineering (7 marks)

The following are guidelines of usability engineering that FOS utilizes:

Minimize memorization:

- FOS has functions of the system packaged as buttons which are clearly labeled with appropriate sizes and easy-to-read font. Since all the buttons are explanatory of what they do, the user does not have to worry about forgetting about the functionality of the system.

- The system saves information for the user so that the user doesn't have to re-enter information such as address, eMail, favourite foods. The user only needs to memorize their login information



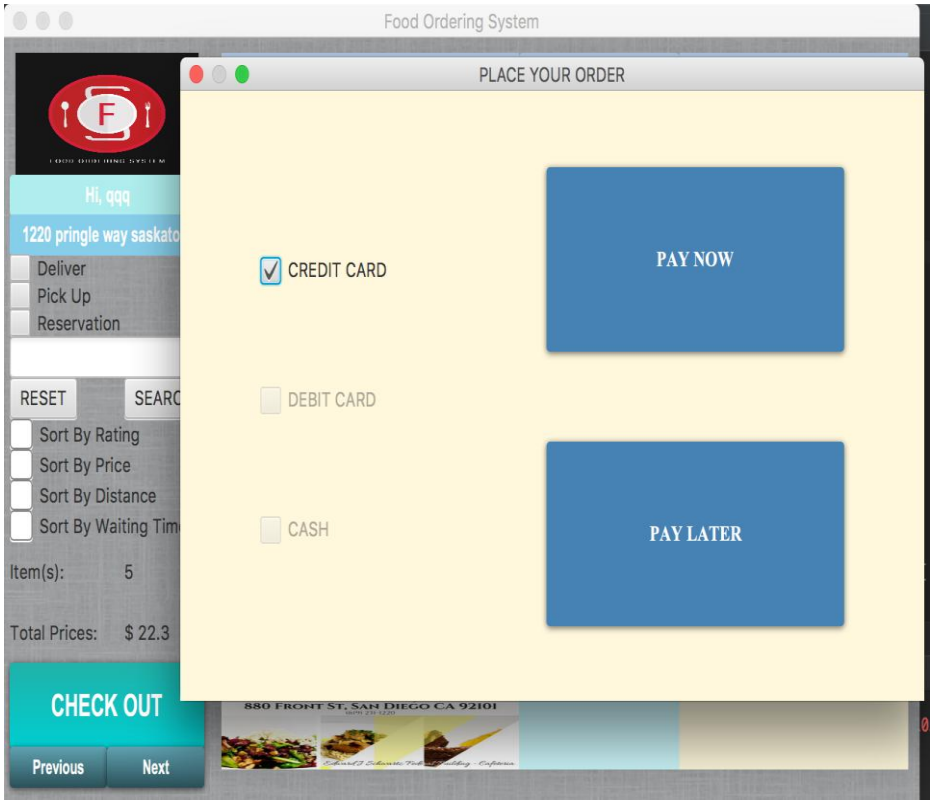
Engineer for Errors:

- FOS employs validation and verification methods to ensure that incorrect input from the user will affect the system negatively. Examples of these include verification of eMail addresses, credit and debit card information, incorrect login information and more. Validation and verification methods are found in the package Payment in classes CVVCodeVerification, CCNumberVerificaiton, DateVerification, creditCardVerification, and in the package Databases package in the class Login.

- Many classes that insert information into the database also have counterpart classes and methods that edit the information entered. This allows the system to easily undo operations. Some examples of these classes are found in the package Database in classes such as EditCustomerInfo, ChangeUserPassword, UpdateWaitingTime, EditLocation, and more.

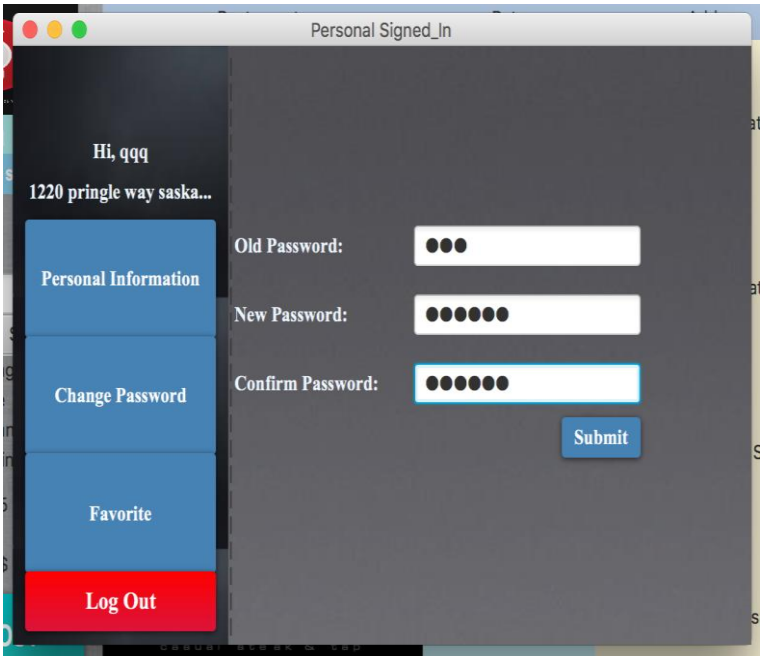
Avoiding Modes:

- Mostly all the functionality of FOS is running under one window. However, the windows that do pop-up are clearly labeled and are concise allowing the user to easily understand the function of the all windows.

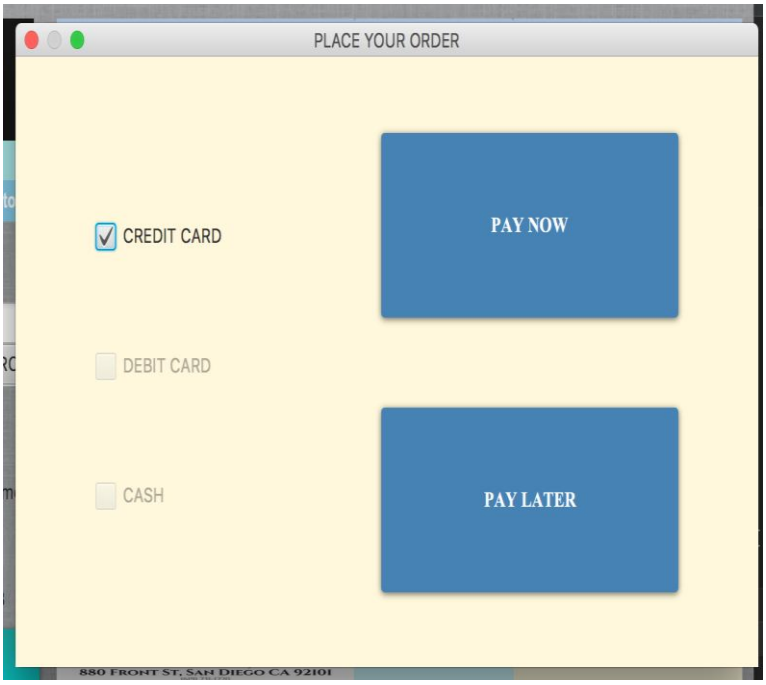


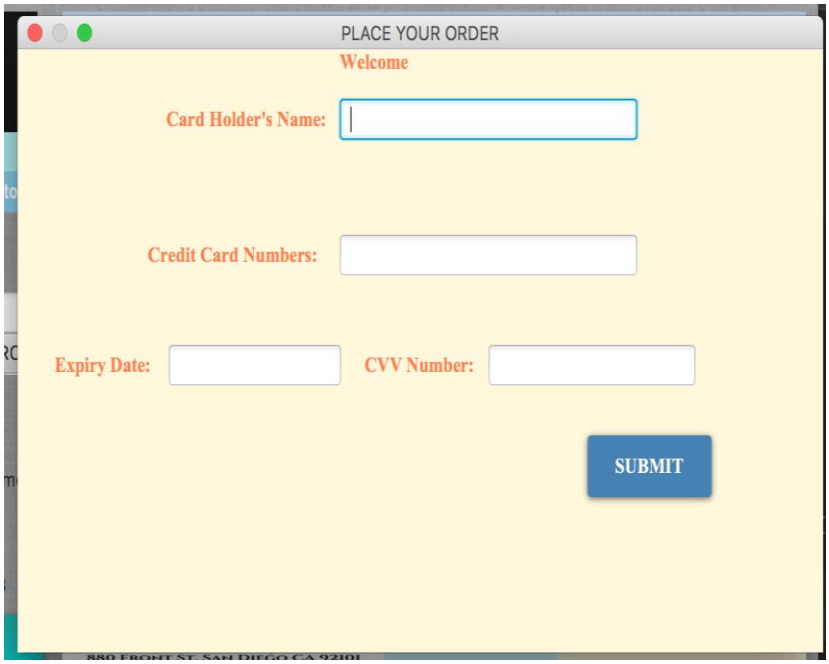
Be Consistent:

- The use of certain colours in the Food Ordering System are consistent throughout the application. The use of different colours on different windows and buttons are used to emphasize to the user about the different functionality of those those components.



- Buttons and windows performing the same functions on the have the same labels and colours to prevent any confusion of the functionality.





5.5 Complete Implementation (15 marks)

Technical Challenges on Implementation:

Learning new Libraries and APIs: The implementation of the system required some coding we were not familiar with prior to coding this application. Members of our group were required to learn and research independently to write a functional implementation. These areas include learning about new libraries and APIs such as simpleJavemail.jar (for sending emails to customers/restaurants), the Google Maps API (for measuring the distance between the restaurant and the customer), and PostGresSQL(for managing the FOS database. Futhermore, not only were these necessary to learn to make a functional program but it was necessary for our group members to learn quickly in order to successfully meet the deadlines for this class.

GRASP (High Cohesion and Low Coupling): It is good practice in computer science, especially for large projects, that different classes should have low coupling and high cohesion. By applying these principles, we prevent modules of code of being dependent on the functionalities of other modules. If too many classes are too dependent, then this creates a problem for all components of code. Furthermore, by having highly focused classes performing similar methods, we create a program that is highly cohesive. The Blazin_Seven members recognized the value of having classes that are very concise even

if there are only a few lines of code in each class. We have tried to implement these principles throughout out program.

Implementing Google Maps API: Our project requires that we calculate the distance between the location of the customer and the location of the restaurant. To achieve this, the Google Maps API was incorporated into the system. However, the use of this API (especially in Java) was a foreign area to everybody in the group. The jargon in this API was quickly learned in order to fulfill a major success scenario in our program.

Writing efficient code: Writing code that is efficient makes the experience of executing the code quickly and doing the required job correctly more easier. However, when incorporating many different modules of code, it was difficult to connect and mediate smooth running between all functionality of the program. This encouraged neat and explanatory code in order for other members of our group to read and understand what has been written. This was time challenging to learn better and more efficient ways to write codes.

Implementing database: The process of implementing a database is a long and strenuous process, we had to learn about some of the functionalities and how to combine it with our java written codes.

Time constraints: Everyone needed to write and finish their codes and make sure they run properly within a time constraint before the submission date. The writing of code was a continuous process because there was need for editing and implementing new ideas or removing irrelevant ideas which made achieving our goals by deadlines even harder to meet. However, this was achieved by having regular meetings to discuss the milestones and revising our goals every meeting to ensure that our program is at its expected stage at each milestone.

5.6 User Manual (7 marks)

Please see attached User Manual

6. Project plan, Budget Justification and Performance Evaluation (REQUIRED + 7 marks)

List of Tasks		**Completed by whom and % Contributions if done in group (provide in number of hours)**	**Comments**
	Abstract	**Ara561/Angu – 0.25 hr**	
1.	**Introduction**	**Ara561/Angu – 0.5 hr**	
2.	**Requirements and Early Design**		

	2.1	**Summary Use Cases**	**Ara561/Angu – 1.25 hr, jak472/Kocur – 1 hr**	
	2.2	**Fully-dressed Use Cases**	**Ara561/Angu – 2 hr**	
	2.3	**Use Case Diagram**	**Ror716/Raji – 1 hr**	
	2.4	**Domain Model**	**Oriade/eoo983 – 1 hr**	
	2.5	**Glossary**	**Oriade/eoo983 – 1.5 hr**	
	2.6	**Supplementary Specification**	**Ror716/Raji – 1hr**	
	2.7	**System Sequence Diagrams**	**Jak472/Kocur(0.5 hr), yid164/Dong (0.5 hr), yur013/Rong (0.5 hr), ara561/Angu(0.5 hr), ror716/Raji(0.5 hr), hal361/Li (0.5 hr), eoo983/Oriade (0.5 hr)**	
	2.8		**Jak472/Kocur (10 min), yid164/Dong (10 min), yur013/Rong (10 min), Aparna (10 min), Ridwan (10 min), hal356/Li (10 min), eoo983/Oriade (10 min)**	
	2.9	**Obtaining User Feedback**	**Jak472/Kocur(0.5 hr), yid164/Dong (0.5 hr), yur013/Rong (0.5 hr), ara561/Angu(0.5 hr), ror716/Raji(0.5 hr), hal361/Li (0.5 hr), eoo983/Oriade (0.5 hr)**	
3.	**Updated Design and Unit Testing**			
	3.1	**System Operations**	**Jak472/Kocur (10 min), yid164/Dong (10 min), yur013/Rong (10 min), ara561/Angu (10 min), ror716/Raji (10 min), hal356/Li (10 min), eoo983/Oriade (10 min)**	
	3.2	**Sequence or Communication Diagrams with GRASP Patterns**	**Jak472/Kocur(1 hr), ara561/Angu (1hr)**	
	3.3	**Class Diagram**	**Ara561/Angu – 1.5 hr**	
	3.4	**Unit Testing**	**Yur013/ Rong – 2 hr**	
4.	**Reengineering and Mutation Testing**			

	4.1	**Code Smells**	**Yur013/ Dong – 2 hr**	
	4.2	**Refactoring**	**Yur013/ Dong – 2 hr**	
	4.3	**Gang of Four Design Patterns**	**Yur013/ Dong – 2 hr**	
5.	**Complete Implementation and Product Delivery**			
	5.1	**Naming Conventions**	**Ara561/Angu – 5 min**	
	5.2	**Commenting**	**Ara561/Angu – 5 min**	
	5.3	**Pretty-printing of the source**	**Yid164/Rong – 0.5 hr**	
	5.4	**Usability Engineering**	**Ara561/Angu, jak472/Kocur – 1.5 hr**	
	5.5	**Complete Implementation**	**Eoo983/Oriade – 0.5**	
	5.6	**User Manual**	**jak472/Kocur – 20 min**	
6.	**Project plan, Budget Justification and Performance Evaluation**		**Ror716/Raji – 0.5 hr, ara561/Angu – 10 min**	
7.	**Conclusion**			
	Acknowledgements		**Ara561/Angu – 10 min**	
	References		**Ara561/Angu – 10 min**	
Total % Contributions (in hours) of the group members			**Ara561/Angu: 10 hrs** **Ror716/Raji: 5 hrs and 40 min** **Yur013/Rong:9 hrs and 20 min** **Jak472/Kocur: 3 hr and 40** **hal356/Li: 1 hr and 20 min** **eoo983/Oriade:3 hr and 50 min** **yid164/Dong: 1 hr and 50 min**	
Total (in hours)			**35 hours and 40 min**	

Group Member	Tasks Responsible For	% Contributions if not done alone, and then say who helped and how much	Comments
Josh	Database Implementation	Spent 20 hours. Received 80 hour of help from Yinsheng on database Implementation	
	Operation Contract	Spent 1 hour	
	Milestone 2,3,4,5 write up and minor tasks	Spent 15 hours. Received hour 3 of help from Aparna	
	System & User Testing	Spent 2 hours	
	Code entities modification	Spent 10 hours	
	Presentation	Spent 1 hour	
	Total for LastName1	28 hours	
Yinsheng	Database Implementation	Spent 80 hours. Received 20 hour of help from Josh on database Implementation	
	Google Map Implementation	Spent 30 hours	
	System Sequence Diagram	Spent 2 hours	
	System Diagram	Spent 3 hours	
	Email Verification	Spent 7 hours. Received 25 hours of help from Yinsheng on Email Verification	
	Operation Contract	Spent 1 hour	
	Search Optimization	Spent 10 hours. Received hours of help from Duke	
	Placing Order	Spent 10 hours	
	Code Refactoring	Spent 6 hours Received 9 hours of help from Ridwan & Eric	

	System & User Testing	Spent 2 hours	
	Presentation	Spent 1 hour	
	Total for LastName2	153 hours	
Duke	Database Search & Sort Functions	Spent 15 hours.	
	System Sequence Diagram	Spent 2 hours	
	System Diagram	Spent 2 hours	
	Operation Contract	Spent 1 hour	
	Code Smells and Refactoring	Spent 18 hour	
	System & User Testing	Spent 2 hours	
	Presentation	Spent 1 hour	
	Total for LastName3	41 hours	
Aparna	Class Diagram	Spent 5 hours	
	Class Diagram	Spent 5 hours	
	Operation Contract	Spent 1 hour	
	System & User Testing	Spent 2 hours	
	Presentation	Spent 1 hour	
	Total for LastName4	53 hours	
	Total hours		
Ridwan	System Diagram	Spent 3 hours	
	Use Case Diagram	Spent 5 hours	
	Project Logo	Spent 1 hour	
	Credit Card Verification	Spent 25 hours	
	User Interface	Spent 7 hours	
	Operation Contract	Spent 1 hour	
	Code Refactoring	Spent 3 hours Received 12 hours of help from Yinsheng & Eric	
	System & User Testing	Spent 2 hours	
	Presentation	Spent 1 hour	
	Total for LastName4	37 hours	
	Total hours		
Eric	System Sequence Diagram	Spent 2 hours	
	System Diagram	Spent 3 hours	
	User Interface	Spent 80 hours.	

		Received 7 hours of help from Ridwan on Credit Card Payment UI	
	Operation Contract	Spent 1 hour	
	Domain Model	Spent 3 hour	
	Code Refactoring	Spent 6 hours Received 9 hours of help from Ridwan & Yinsheng	
	System & User Testing	Spent 2 hours	
	Presentation	Spent 1 hour	
	Total for LastName4	98 hours	
	Total hours		
Emmanuel	Glossary	Spent 3 hours.	
	Email Verification	Spent 25 hours. Received 7 hours of	
	Operation Contract	Spent 1 hour	
	System & User Testing	Spent 2 hours	
	Presentation slides	Spent 8 hour	
	Milestone 6 tasks	Spent 15 hours	
	Milestone 3,4 and 5 tasks	Spent 15 hours	
	Total for LastName4	46 hours	
	Total hours		

7. Conclusion (1 mark)

The Blazin_Seven members, Aparna Angu, Duke Rong, Emmanuel Oriade, Eric Li, Josh Kocur, Ridwan Raji, and Yinsheng Dong would like to thank Professor Chanchal Roy for conducting Computer Science 370 imparting knowledge about the software engineering project processes and encouraging students on their projects. We also would like to thank the Teaching Assistants for providing guidance in our project process and providing suggestions to improve our overall project experience.

References (1 mark)

Reference:

[1] The JUnit website: http://www.junit.org/

[2] Chanchal Roy and Jim Cordy’s Technical Report: http://research.cs.queensu.ca/TechReports/Reports/2007-541.pdf

[3]: PMD plugins: https://pmd.github.io/#plugins (all IDEs)

[4]: FindBugs for IntelliJ https://plugins.jetbrains.com/plugin/3847-findbugs-idea4

[5]: CheckStyle for IntelliJ https://plugins.jetbrains.com/plugin/1065-checkstyle-idea

[6]: LucidCharts for Diagrams https://www.lucidchart.com/

[7]: Google Maps API https://github.com/googlemaps/google-maps-services java/tree/master/src/main/java/com/google/maps