



Course Code : CST209
Course Name : Object-Oriented Programming-C++
Lecturer : Geetha Kanaparan
Academic Session : 2024/04
Assessment Title : Final Project
Submission Due Date : 15/7/2024

Prepared by :

Student ID	Student Name
AIT2209934	Ang Yi Xun

Date Received : _____

Feedback from Lecturer:


Mark:

Own Work Declaration

I/We acknowledge that my/our work will be examined for plagiarism or any other form of academic misconduct, and that the digital copy of the work may be retained for future comparisons.

I/We confirm that all sources cited in this work have been correctly acknowledged and that I/we fully understand the serious consequences of any intentional or unintentional academic misconduct.

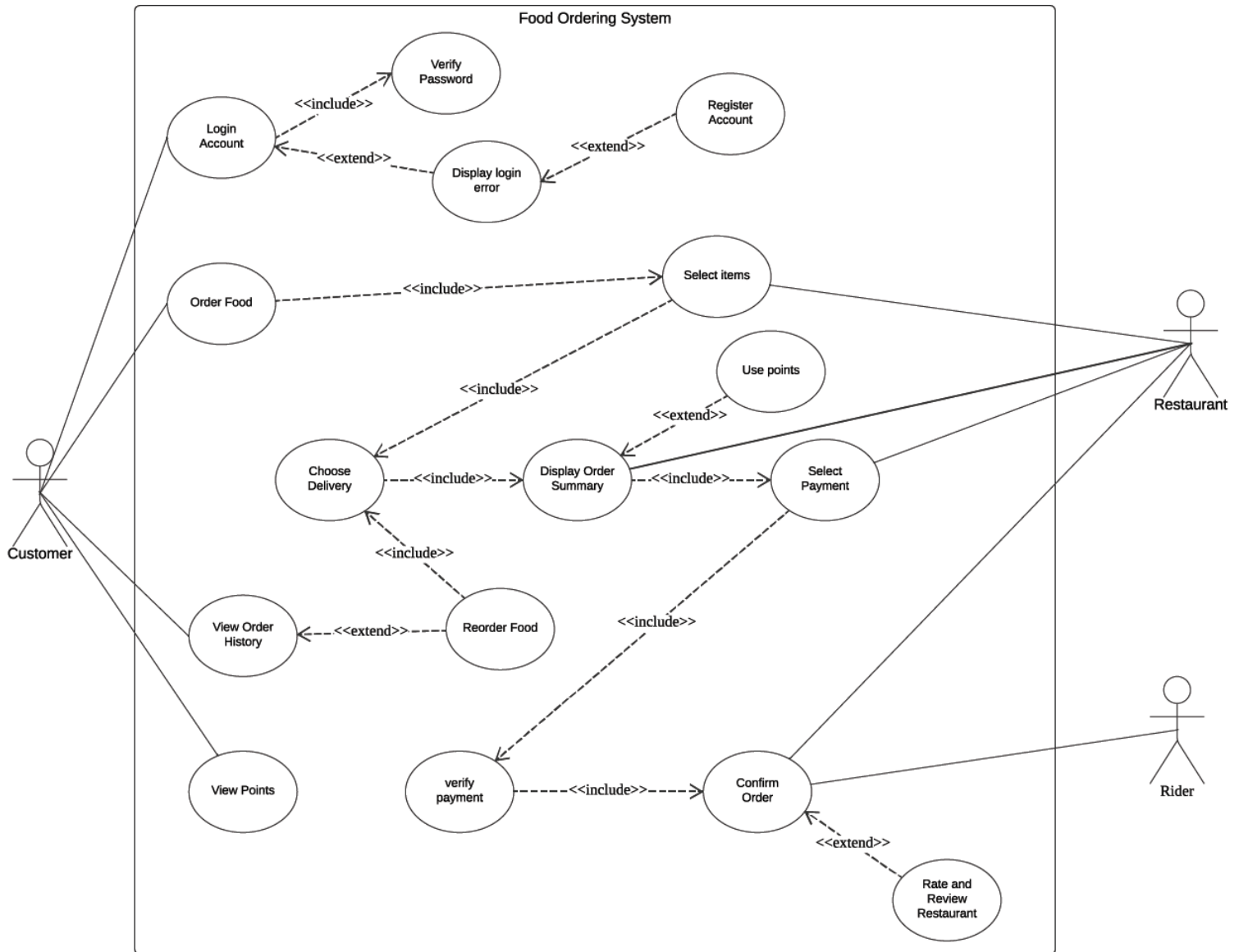
In addition, I/we affirm that this submission does not contain any materials generated by AI tools, including direct copying and pasting of text or paraphrasing. This work is my/our original creation, and it has not been based on any work of other students (past or present), nor has it been previously submitted to any other course or institution.

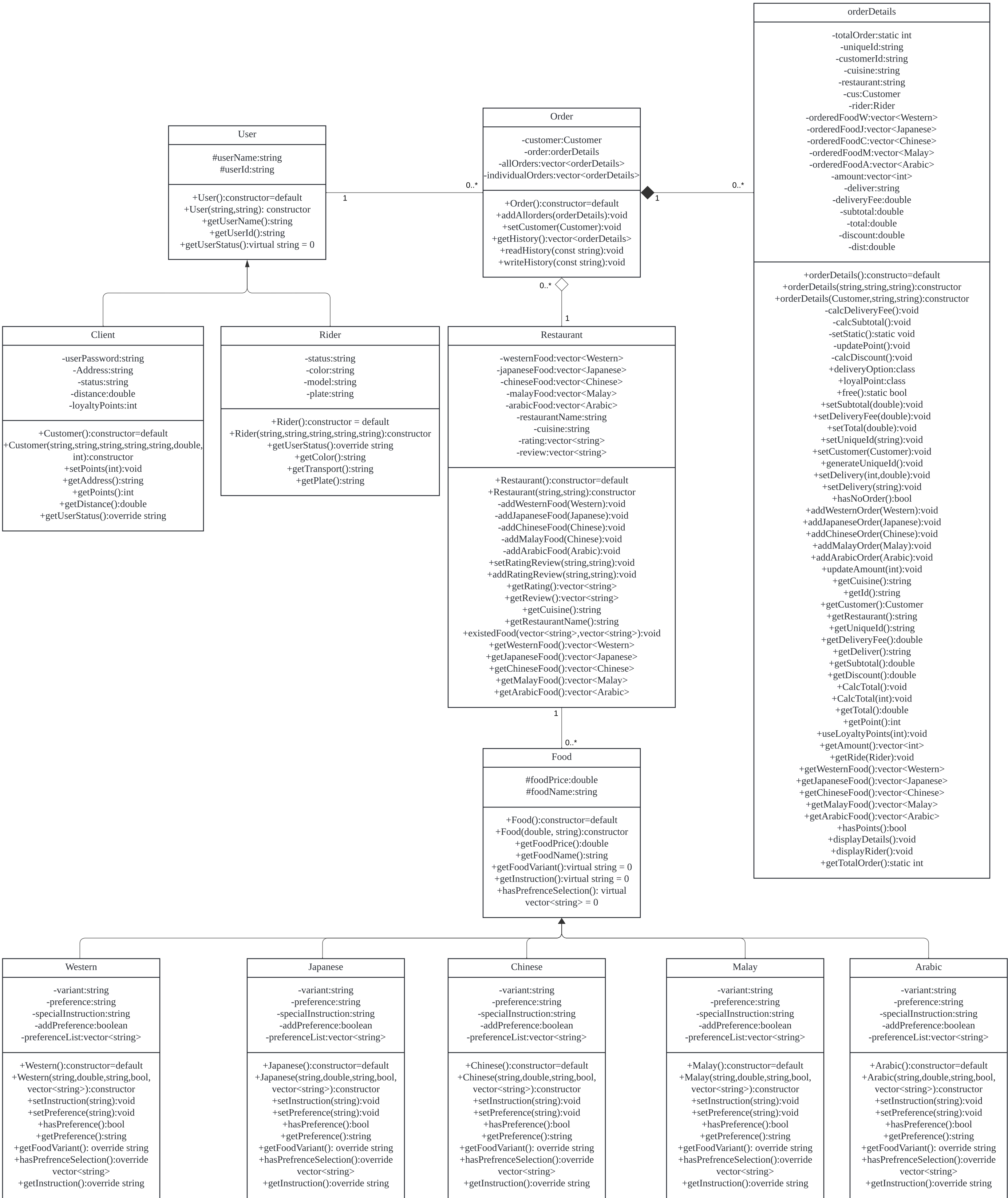
Signature: 

Date: 11/7/2024

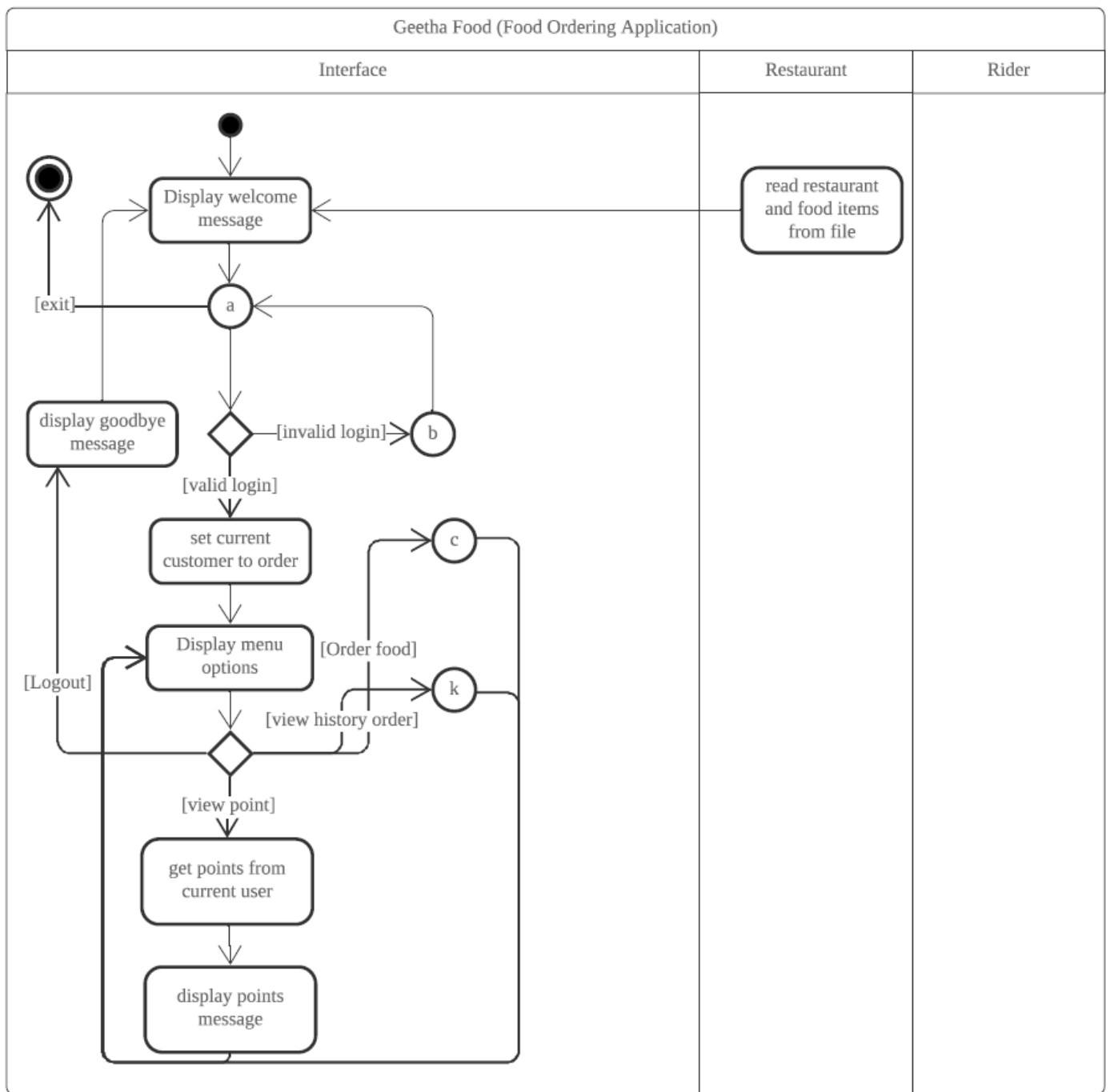
UML Diagrams

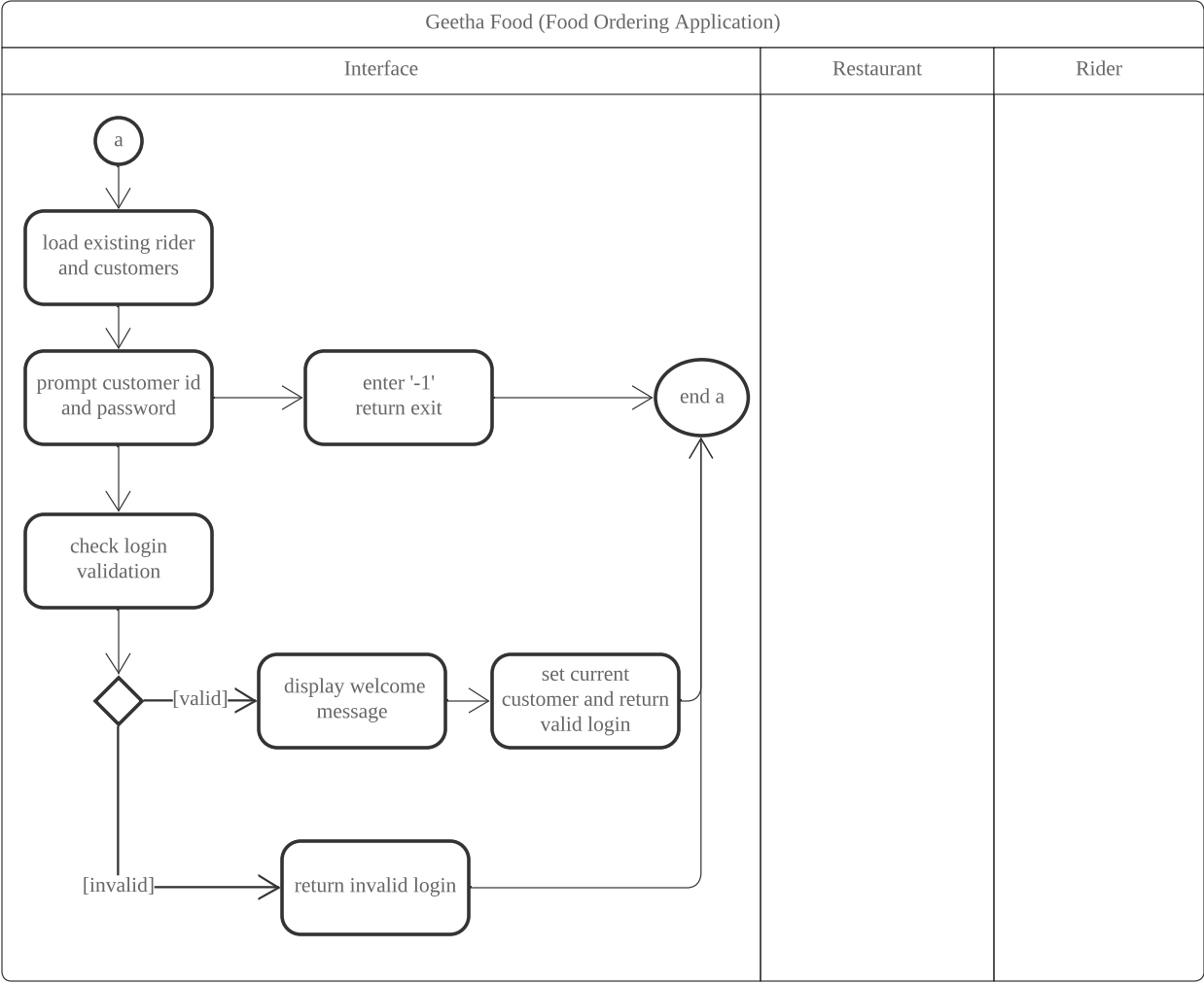
Use Case Diagram

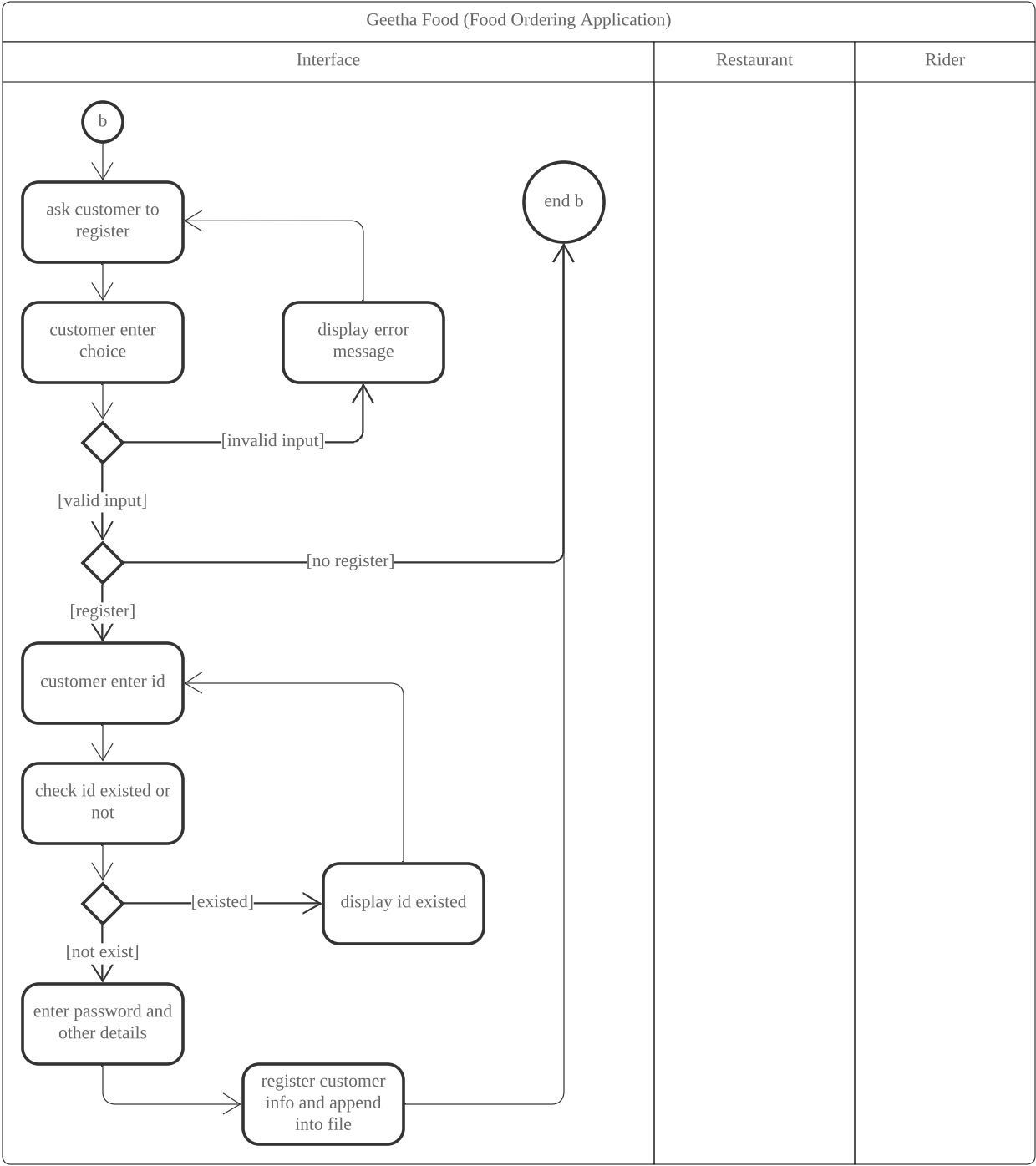


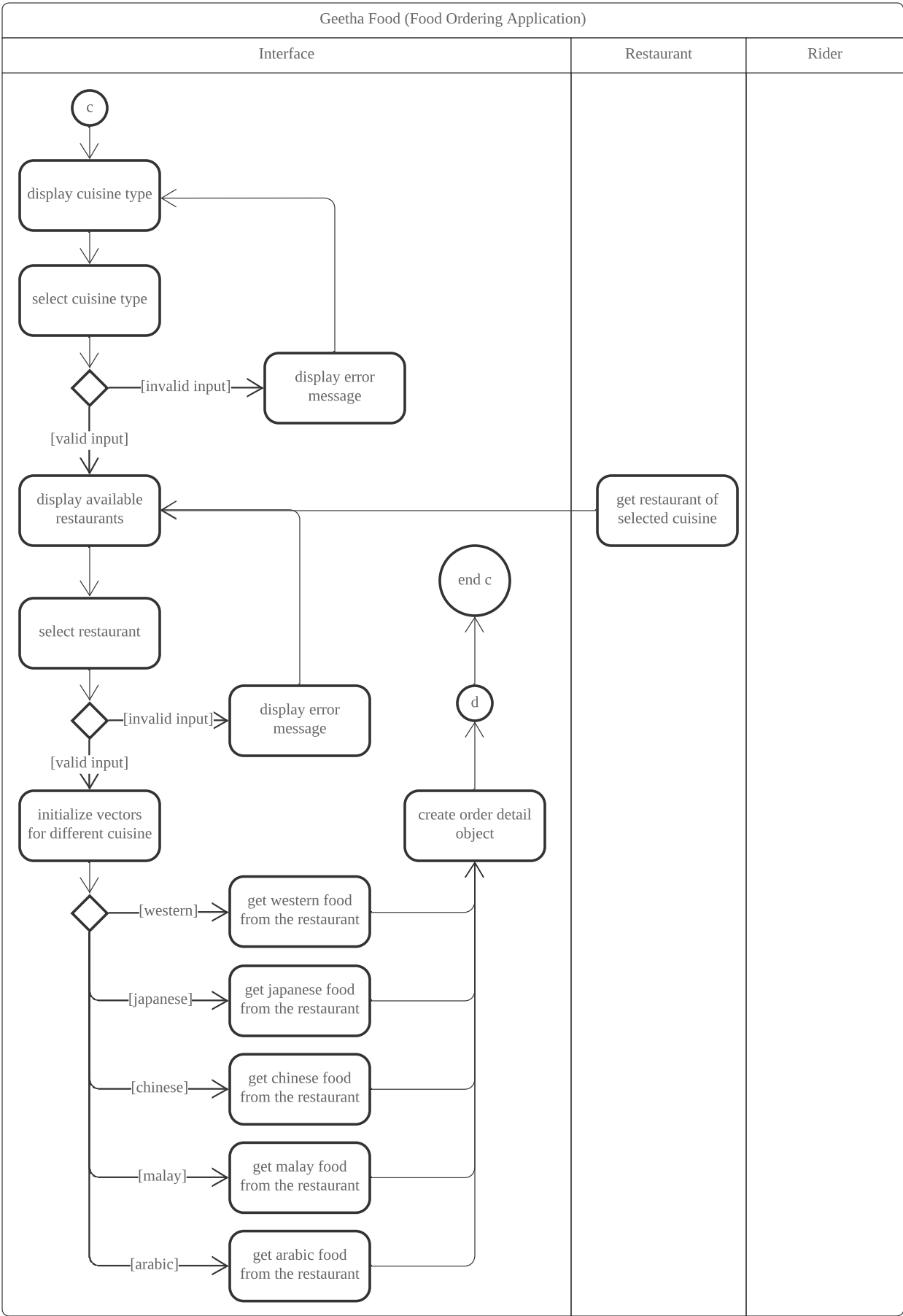


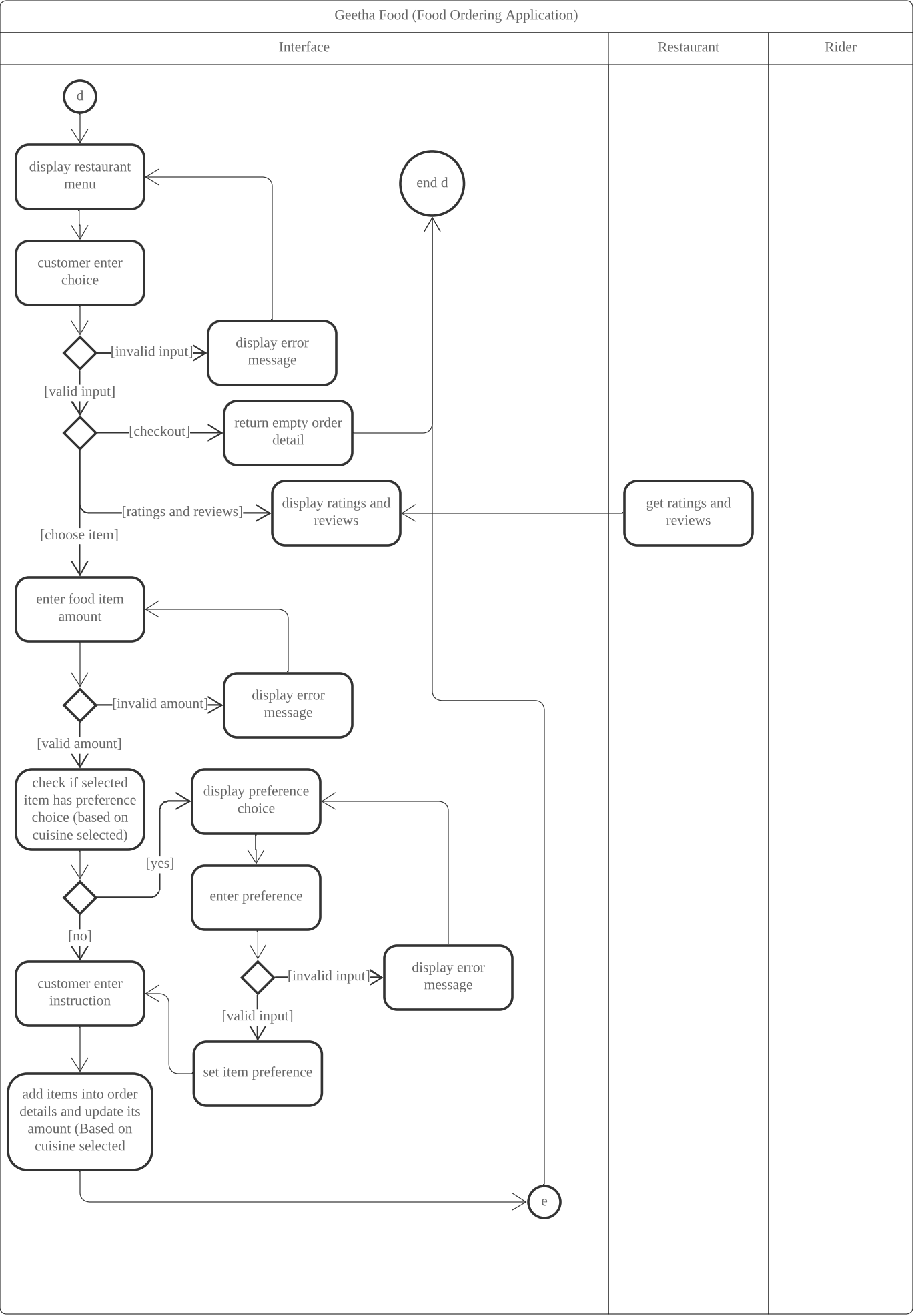
Activity Diagram

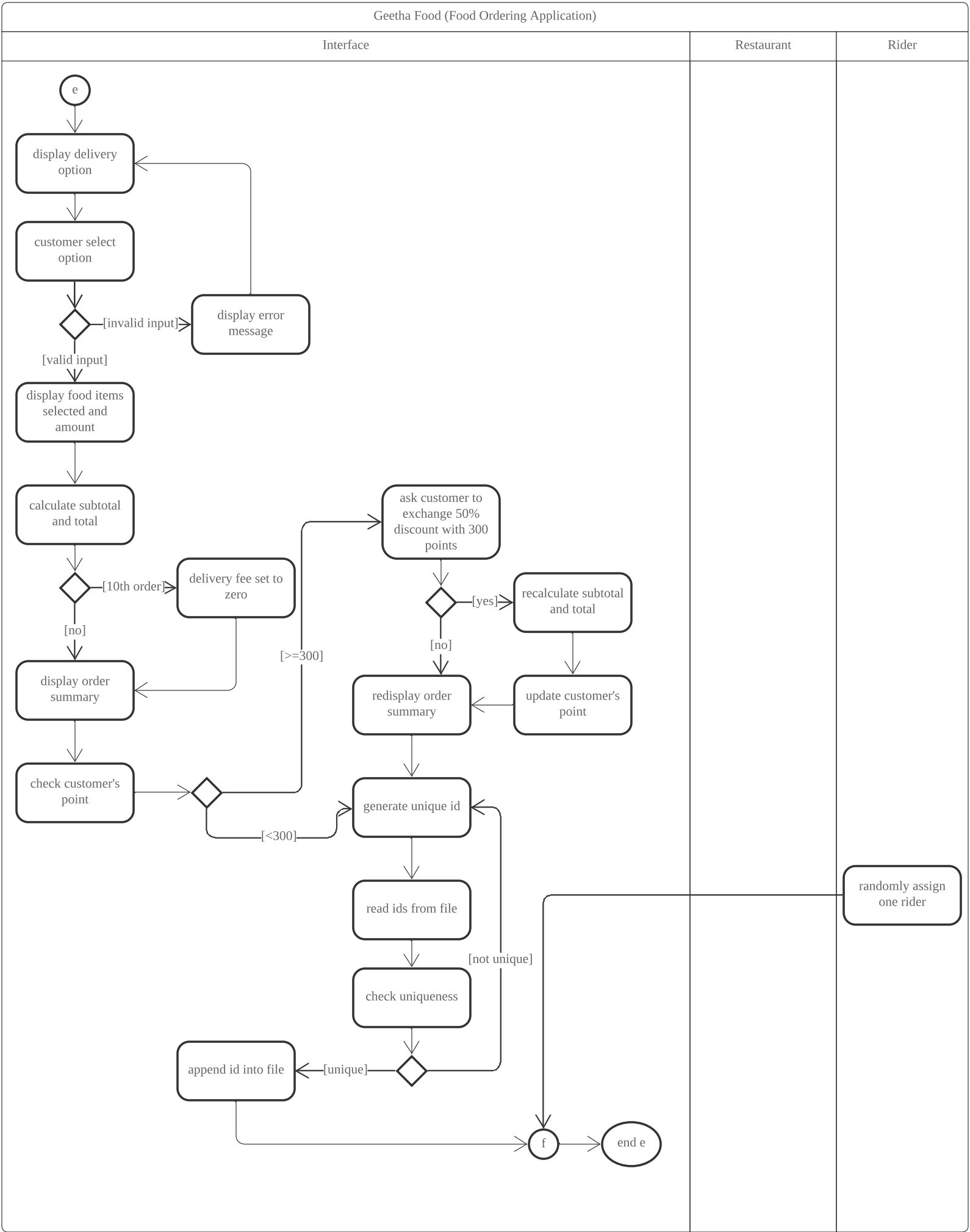


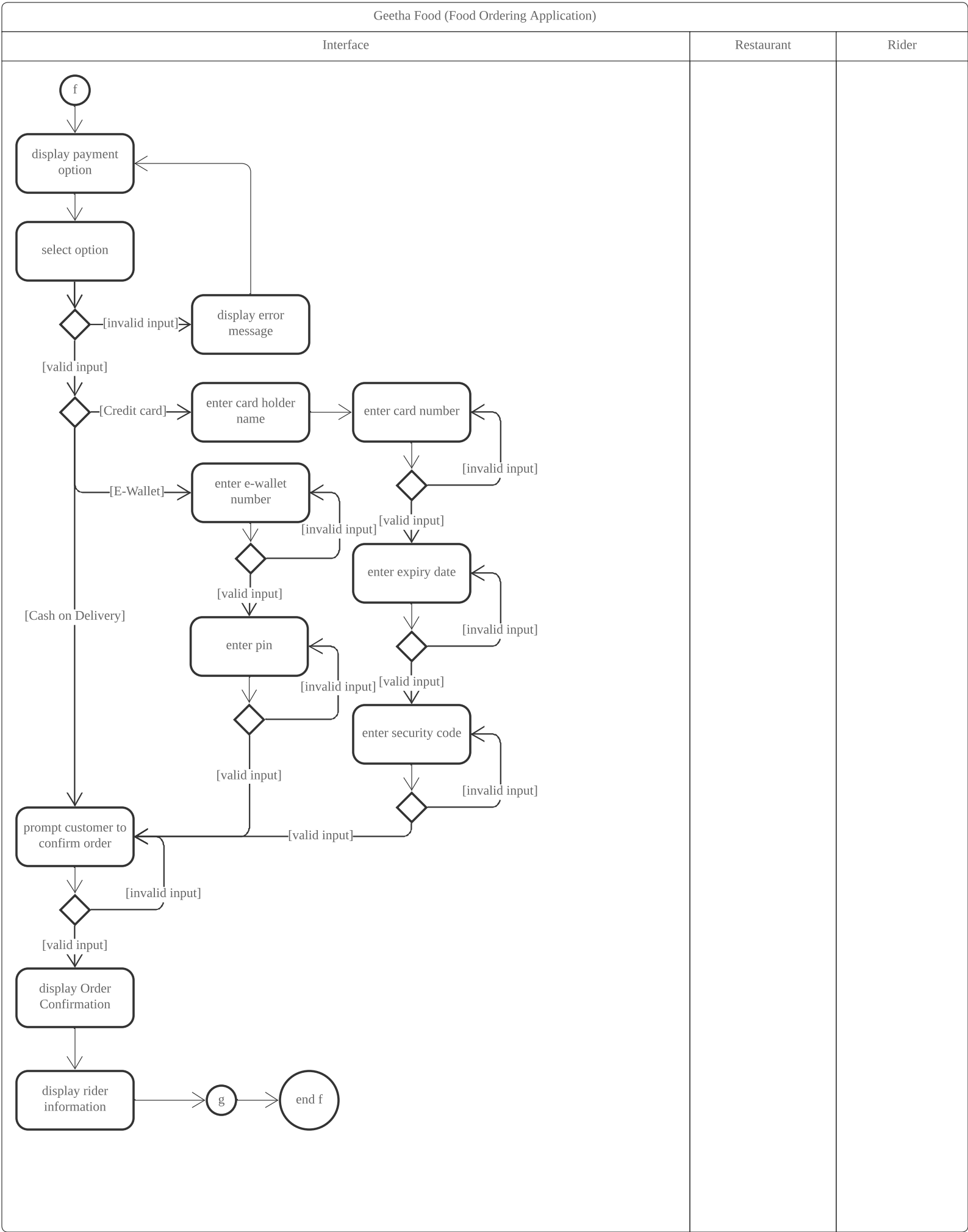


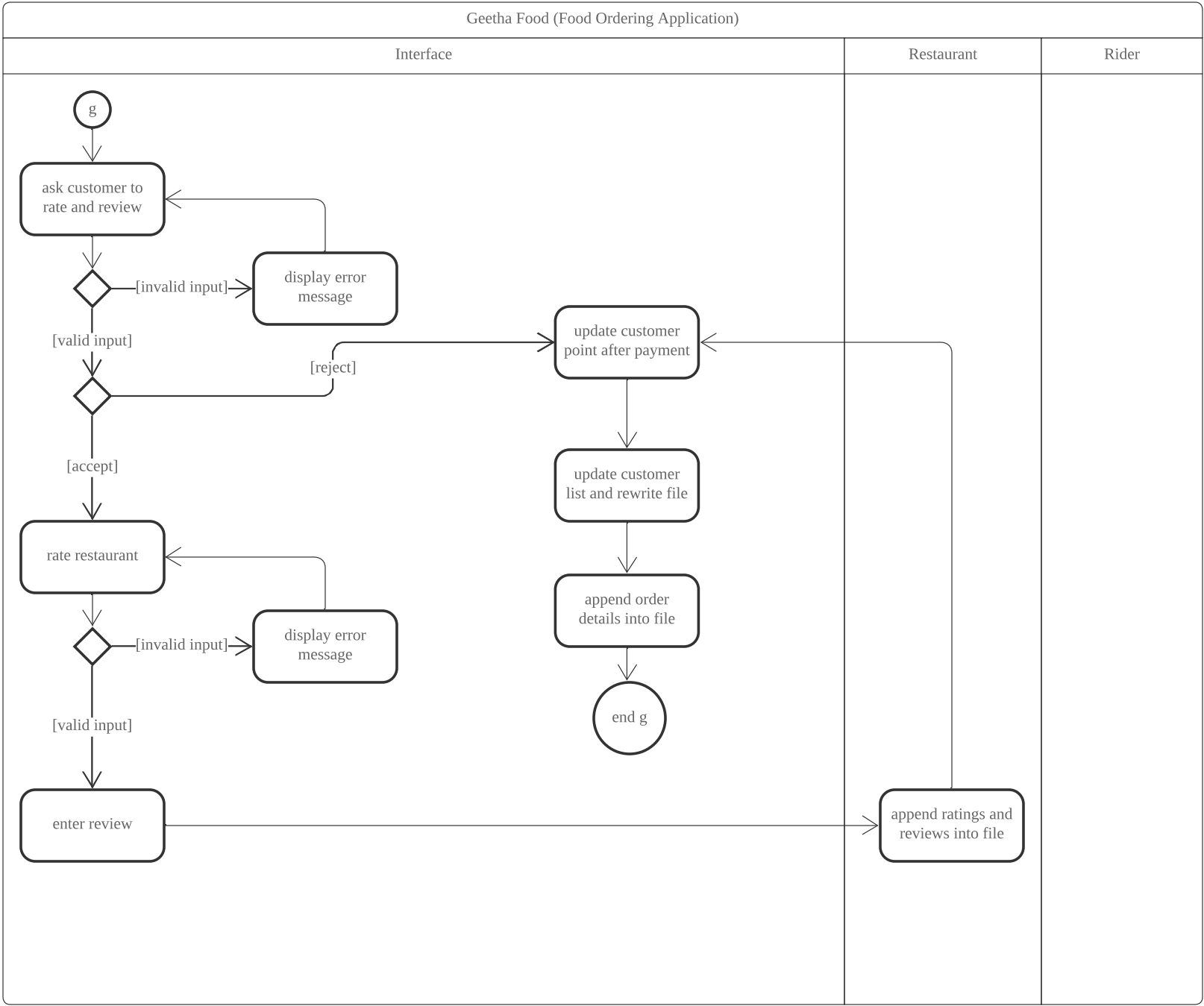


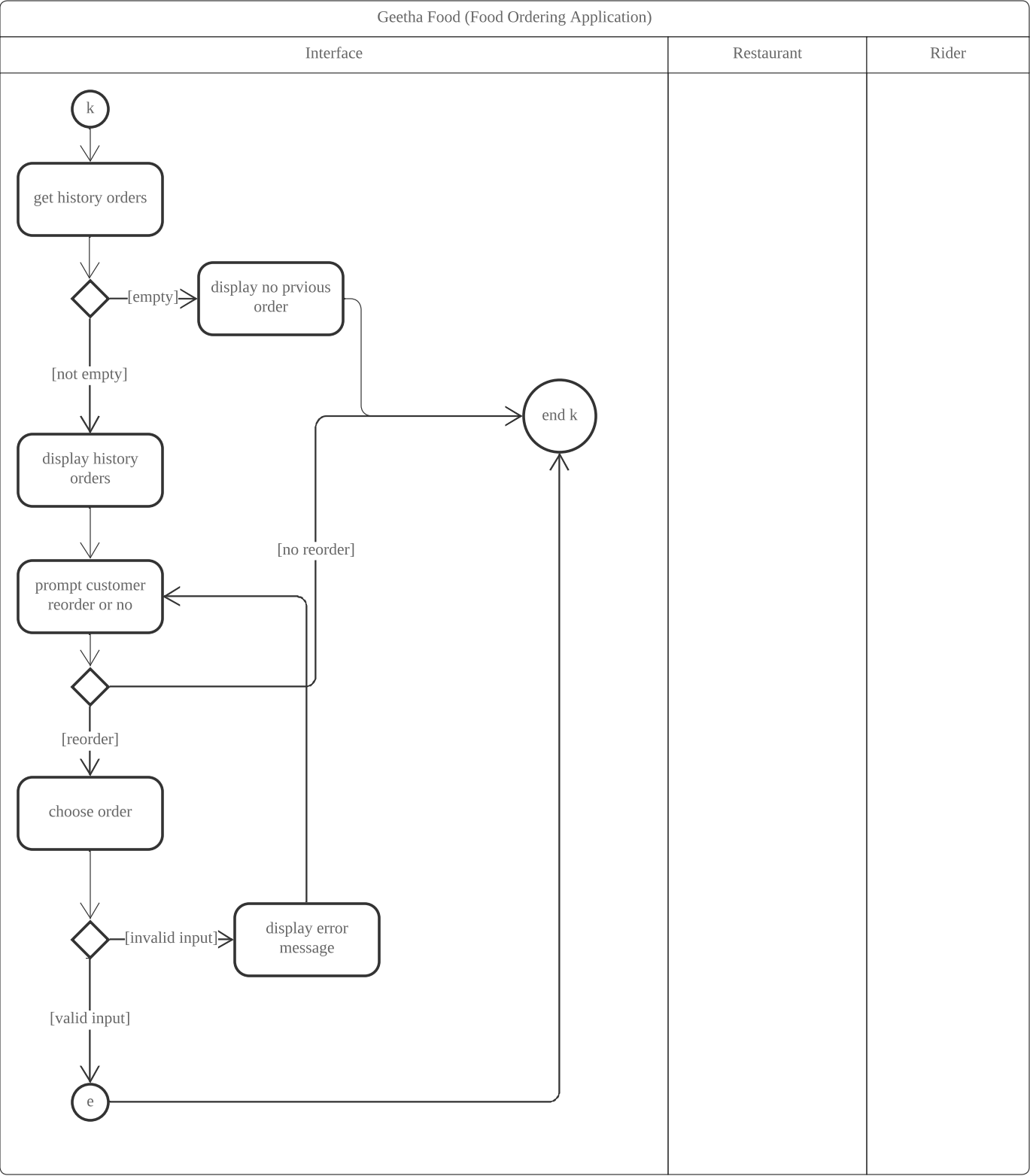












STL Explanation

Standard Template Library (STL) is being utilized to ease the development process of a c++ program with suitable data structures and methods to search, sort and manipulate data. In this c++ final project, I have made use of 2 STLs which are vector and algorithm.

Vector

Vector is a dynamic array which provide sequential approach in inserting or retrieving. In this project, I use vectors as containers as to store different objects like food items or previous orders.

Firstly, I have used vector when reading food items' details needed to construct each food object from respective files.

```
foodInfo readCSV_FOOD(const string& filename){
    ifstream file(filename);
    vector<vector<string>> details;
    vector<vector<string>> preList;

    if (!file.is_open()) {
        cerr << "Could not open the file: " << filename << endl;
        foodInfo Details = {};
        return Details;
    }
    string line;
    while (getline(file, line)) {
        stringstream ss(line);
        string value;
        vector<string> values;
        vector<string> pre;
        int i = 1;
        while (getline(ss, value, ',')) {
            if(i<5)
                values.push_back(value);
            else
                pre.push_back(value);
            ++i;
        }

        details.push_back(values);
        preList.push_back(pre);
    }
    foodInfo Details = {details, preList};
    file.close();
    return Details;
}
```

Figure 1: vector used in extracting food item's details

From this figure 1, we can see proper usage of vector of data type string to first store the information of the first food item such as its name, price and preference choice, if any into the vector of string sequentially. Note that we have 2 string vectors used when in the while loop of getline as we have to store the preference choice as a vector before passing it as argument to construct a food object. These vectors are then push back into vectors which store vector of strings whereby each vector of strings contain the details of a specific food item. The details can extract from a single file using a

vector which then can be pass back for constructing objects by iterate through the vectors and use elements inside.

```
vector<string> readUniqueId(const string& filename){
    ifstream file(filename);
    vector<string> values;
    if (!file.is_open()) {
        cerr << "Could not open the file: " << filename << endl;
        return values;
    }
    string line;
    if(getline(file, line))
    string line;
    while (getline(file, line)) {
        stringstream ss(line);
        string value;
        while (getline(ss, value, ',')) {
            values.push_back(value);
        }
    }

    file.close();
    return values;
}
```

Figure 2: Vector to store unique Ids

In figure 2, We use vector to simply store any generated id to prevent repetition and ensure uniqueness of Id generation afterwards.

```
vector<Customer> rewriteCustomer(const string& filename, Customer c, vector<Customer> customerList,
    for(auto& cus : customerList){
        if(c.getUserId() == cus.getUserId()){
            cus.setPoints(c.getPoints());
        }
    }
    ofstream file(filename, ios::out);
    if (!file.is_open()) {
        cerr << "Could not open the file: " << filename << std::endl;
        return customerList;
    }
    file << "Users";
    file << "\n";
    for(auto& cus : customerList){
        file << cus.getUserName();
        file << ",";
        file << cus.getUserId();
        file << ",";
        file << cus.getUserStatus();
        file << ",";
        file << cus.getUserPassword();
        file << ",";
        file << cus.getAddress();
        file << ",";
        file << cus.getDistance();
        file << ",";
        file << cus.getPoints();
        file << "\n";
    }
}
```

Figure 3: Store vector of Customers

Besides, we use vector to store customers and pass the vector of customers around for writing into file or detail comparison to identify the customer exists or is valid. From

figure 3 we can see that a vector of customers is pass in as argument in this writeCustomer function (update each customer's point and save into file). The function first iterates through the customer vector to search for the current logged in customer and update the points after ordering. Note that the vector of customers allows the function to access information of each customer and rewrite them into a file by a for loop. Thus, the vector serves well in passing information of customers by holding the customers instead of having multiple string variables or string arrays to store the details before passing them into the function.

```
void checkRegisterId(string id, vector<Customer> customerList){
    string msg = "Id used.";
    for(auto& ctm : customerList){
        if(id == ctm.getUserId()){
            throw msg;
        }
    }
}

bool checkValidCustomer(string id, string pw, vector<Customer> customerList){
    for(auto& customer : customerList){
        if(id == customer.getUserId() && pw == customer.getUserPassword()){
            return true;
        }
    }
    cout << endl;
    cout << "Invalid login details, please try again.";
    cout << endl << endl;
    return false;
}
```

Figure 4: vector of customers usage

Figure 4 shows similar usage is implemented when we want to check if the customer registers a new account, the application will be able to check if the customer's new id is unique by comparing it through all customers' id. The same goes to login function to check whether the entered id and password matches any existing customers.


```
vector<Restaurant> tempResList;
if(cin.fail()){
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    string msg = "Invalid input";
    throw msg;
}
string term;
switch(choice){
    case 1:
        term = "western";
        break;
    case 2:
        term = "japanese";
        break;
    case 3:
        term = "chinese";
        break;
    case 4:
        term = "malay";
        break;
    case 5:
        term = "arabic";
        break;
    default:
        string msg = "Out of Range";
        throw msg;
        break;
}
for(auto& res : restaurantList){
    if(term == res.getCuisine()){
        tempResList.push_back(res);
    }
}
```

Figure 5: vector to get selected cuisine restaurants

Another implementation of vectors is to return a vector of restaurants which serves the type of cuisine selected by the customers. In figure 5, the function will get a vector of restaurants which stores all existing restaurant of all cuisine and pass the restaurant of specific cuisine type into a temporary vector and return to the main function for display purpose. This way, the restaurants can be pass or technically create a new one and redefine them in another vector which will be use for other purposes. As a result, no changes will be made towards the original vector which stores the restaurants as restaurants do not change during the ordering process.

```
vector<Customer> customerList;
vector<Rider> riderList;
vector<Restaurant> restaurantList;
vector<Restaurant> tempResList;
```

Figure 6: vectors created when code runs

In the main function, vectors such as customerList, riderList, restaurantList and tempResList are initialized to store existing customers, riders and restaurants after reading their details and construct the objects. The storage of all the objects in vectors accelerate the process of accessing information from specific objects by simply iterate and compare. The usage of vectors provides simple and efficient storing method which suits the requirement of this final project.

Algorithm

In this final project, we are required to do exception handling to handle invalid inputs from the customers. It is easy to deal with problems such as input out of range, check if string entered is digit, or even if the input from the customer matches the requirement in order to proceed to the next steps shown in figures below.

```
if(n<min || n>max){
    string msg = "Out of range";
    throw msg;
}
```

Figure 7: input range validation

```
bool toRegister(char reg){
    if(reg == 'Y' || reg == 'y'){
        return true;
    }
    else if(reg == 'N' || reg == 'n'){
        return false;
    }
    else{
        string errmsg = "Invalid input";
        throw errmsg;
    }
}
```

Figure 8: Input validation for registration

```
void ratingValidtion(string number){
    string msg = "Invalid input";
    for(int i = 0; i<number.size(); i++){
        if(!isdigit(number[i])){
            throw msg;
        }
    }
    if(stoi(number) < 1 || stoi(number) > 5)
        throw msg;
}
```

Figure 9: Input validation for string hold digits

However, when it comes to dealing with integer input, the code will return error if a none digit character is entered. Thus, we will have to deal with the errors by first testing if cin gets the correct type of data and deal with buffers of the input.

```
//check input validation
void digitInputValidation(int n, int min, int max){
    // (Singh, 2016)
    // check if inputs are digit (Pai, 2015)
    if(cin.fail()){
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        string msg = "Please enter digits.";
        throw msg;
    }
    if(n<min || n>max){
        string msg = "Out of range";
        throw msg;
    }
}
```

Figure 10: integer input validation

From figure 10, we use `cin.fail` to test if `cin` gets the proper data type, if not the function will clear previous input and buffers before displaying the error message. This is where algorithm is being utilized. `Max` function is use along with `numeric_limits` to gets the maximum possible value of certain type, it is `streamsize` representing the `cin` or customer input for this case. It helps by applying `ignore` on every buffer character until the character new line ‘\n’ is met or reach maximum finite value (scorchingeagle, 2021). As a result, the input buffers will be cleared which prevents the code from catching errors.

The STL of algorithm is being used to deal with exception handling on customer’s input. It is important as it ensures customer enter the correct input with the correct data type in order for the application to work well while at the same time allows application to be user friendly as customers will know what is wrong with their input instead of exiting the application showing errors leading customers to confusion.

Inheritance and Polymorphism Explanation

Inheritance

In this project, I have 2 inheritances implemented. First is going to be Food class as parent class and its child class will be Western class, Japanese class and so on which represents different cuisine type. As for the second example is the User class along with its derived class of Customer class and Rider Class. Example of codes in figures below.

```
class Food{
protected:
    double foodPrice;
    string foodName;
public:
    Food() = default;
    Food(double price, string name){
        foodPrice = price;
        foodName = name;
    }

    double getFoodPrice(){
        return foodPrice;
    }

    string getFoodName(){
        return foodName;
    }
    // pure virtual function
    virtual string getFoodVariant() = 0;
    virtual vector<string> hasPreferenceSelection() = 0;
    virtual string getInstruction() = 0;
};

class Western : public Food{
private:
    string variant;
    string preference;
    string specialInstruction;
    bool addPreference;
    vector<string> preferenceList;
public:
    Western() = default;
    Western(string name, double price, string var, bool aP, vector<string> listPrefer = {}):
        Food(price, name){
        variant = var;
        addPreference = aP;
        preferenceList = listPrefer;
    }
};
```

Figure 11: Inheritance (Food as parent class, Western as child class)

```

class User{
protected:
    string userName;
    string userId;
public:
    User() = default;
    User(string name, string id){
        userName = name;
        userId = id;
    }
    string getUserName(){
        return userName;
    }
    string getUserId(){
        return userId;
    }
    // pure virtual function
    virtual string getUserStatus()=0;
};

class Customer : public User{
private:
    string userPassword;
    string Address;
    string status;
    double distance;
    int loyaltyPoints;
public:
    Customer() = default;
    Customer(string name,
             string id,
             string stats,
             string password,
             string address,
             double dist, int point):User(name,id){
        userPassword = password;
        Address = address;
        distance = dist;
        status = stats;
        loyaltyPoints = point;
    }
};

```

Figure 12: Inheritance (User as parent, Customer as child)

The purpose of implementing inheritance on different cuisine type class inherit from the Food class is that all cuisine are generally food with its own name and price. Thus, the cuisine class such as Western, Chinese and Malay class are inherited from the food class where they all have their respective names and price. However, they hold different cuisine type which differentiate themselves from each other. The idea is similar when implement inheritance on User class. Both customers and riders have names and ids, and they should both generally be users. As a result, implementation of inheritance on Food and User class is to allow reusing of code or in another words, to prevent redundancy such as Rider class and Customer class are able to use `getUserName()` function which already existed in the base class of User.

Second, inheritance is used to visualize the hierarchy relationship of classes. From the view of food ordering application, it will have different users. The users could specifically be customers who order food items or riders who deliver orders to the customers. This is mainly to represent the real-world relation.

Furthermore, for example inheritance of Customer class and Rider class from the parent class of User also ensure encapsulation and specification which a derived class can have its own specific behaviour while at the same time maintain its general characteristics inherited from its parent class. To further elaborate this, Customer class have specific methods such as get customer's address, points and account password. As for Rider class, it has its own specification such as get deliver transport name, color and car plate. While both classes have their own methods and functions, they both retain their basic function of getting name and id inherited from the base class.

Inheritance provides polymorphism which we will discuss later. All in all, inheritance implemented in this project provides multiple advantages. To further justify if a inheritance is suitable or logical, we could simply just use the is-a relationship. For example, western, chinese, arabic and others cuisine is a food. Customer is a user and rider is also a user. Thus, the usage of inheritance in this project does make sense and it helps a lot in reduce repeating code lines.

Polymorphism

The first polymorphism used in this project is function overloading. Meaning the there will be 2 functions with same name, but they perform differently according to the type of arguments they receive. For example, in orderDetails class it has two 2 setDelivery function, one accepts a int and a double data type paramters while the other accepts only a string argument. Both functions are design to set the delivery option to whether it is direct, standard or saver for that particular order. However, the function that receive string argument is to set delivery option when creating order retrieve or read from file where as the other function is creating new order and it receives customer's input choice and access distance of the restaurant to the customer's address for the purpose of calculating delivery fee. This enable polymorphism which overloads functions for different procedures according to different parameters. Below shows the figures of 2 functions overloading.

```
void setDelivery(string d) {
    deliver = d;
}
```

Figure 13: function to set delivery option based on past orders

```
void setDelivery(int n, double distance) {
    if(cin.fail()) {
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        throw deliveryOption();
    }
    if(n>3 || n<1) {
        throw deliveryOption(n);
    }

    if(n==1)
        deliver = "direct";
    else if(n==2) {
        deliver = "standard";
    }
    else if(n==3) {
        deliver = "saver";
    }
    dist = distance;
    totalOrder += 1;
    ofstream file("total purchase index.csv", ios::out);
    if (!file.is_open()) {
        cerr << "Could not open the file: " << "total purchase index.csv" << std::endl;
        return;
    }
    file << "idx";
    file << "\n";
    file << totalOrder;
    file << "\n";

    file.close();
    calcDeliveryFee();
}
```

Figure 14: overloading function to set delivery option based on customer's choice

Another function overloading being implemented for calculating the total price of the customer's order. If the customer choose to redeem 50% discount using 300 points, the function will receive integer parameter which the integer value indicate the customer agree the redeem and apply 50% discount when calculating the final total price. If there is no redeem being made, the function with no argument is called which it simply calculates the total price without adding any discount to it. The code can be seen from figure below.

```
// calculate total
void CalcTotal() {
    total = subtotal + deliveryFee;
}
// calculate total if 50% discount applied using 300 points
void CalcTotal(int c) {
    total = (subtotal + deliveryFee) * discount;
}
```

Figure 15: Function overloading in calculating the total order price

Next, we facilitate polymorphism by using an abstract class which acts as an interface which a derived class must implement. In this project, both parent class of Food and User are abstract class as both have pure virtual functions where the derived class must implement. For example, in Food class it has abstract function of `getInstruction` and `hasPreferenceSelection()` where the child class must implement. We can visualize the implementation from below figures.

```
virtual vector<string> hasPreferenceSelection() = 0;
virtual string getInstruction() = 0;
```

Figure 16: abstract function in Food class

```
// if has preference choice, return the list of choice
vector<string> hasPreferenceSelection() override{
    if(addPreference){
        return preferenceList;
    }
}
// get special instruction of food item
string getInstruction() override{
    return specialInstruction;
}
```

Figure 17: Implementation of abstract function

```
virtual string getUserStatus()=0;

string getUserStatus() override {
    return status;
}
```

Figure 18: Implementation of abstract function in User class

However, simply having a child class that provide implementation of the abstract function does not mean polymorphism. Polymorphism appears when there is a pointer to the base class which refers to its child class that shows implementation. Thus, we enable the polymorphism by following code shown in figure 19 where `f` which pointed to base class of Food class has a reference of the food which allows it to access special instruction being requested by the customers during order of food items.

```
Food* f = &food;
cout << "Special Instruction: " << f->getInstruction() << endl;
```

Figure 19: pointer to abstract class that refers to derived class that provides implementation

Reference

- Pai, A. (2015, February 11). *Validating user input in C++* - Akshay Pai. HackerEarth.
<https://www.hackerearth.com/practice/notes/validating-user-input-in-c/>
- scorchingeagle. (2021, January 25). *std::numeric_limits::max() and std::numeric_limits::min() in C++*. GeeksforGeeks.
https://www.geeksforgeeks.org/stdnumeric_limitsmax-and-stdnumeric_limitsmin-in-c/
- Singh, M. (2016, June 22). *Clearing The Input Buffer In C/C++*. GeeksforGeeks.
<https://www.geeksforgeeks.org/clearing-the-input-buffer-in-cc/>

APPENDIX 1

MARKING RUBRICS

Component Title	Part 1: Food Ordering Application					Percentage (%)	35
Criteria	Score and Descriptors					Weight (%)	Marks
	Excellent (5)	Good (4)	Average (3)	Need Improvement (2)	Poor (1)		
Classes and objects	Classes are well organised and implemented with multiple header files and if necessary multiple CPP files.	Classes and objects are fully implemented with the separation of class specification implementation.	Classes and objects are fully implemented.	Partial Implementation of classes and objects.	Classes and object were not implemented or is not used at all.	15	
Inheritance	Excellent implementation of Inheritance with base classes.	Adequate implementation of inheritance.	Minimal inheritance implemented.	Partial inheritance implemented.	No inheritance seen in the code.	10	
Polymorphism	Excellent implementation of polymorphism with abstract classes.	Adequate implementation of polymorphism that covers function overloading and operator overloading.	Minimal implementation of polymorphism with some function and operator overloading.	Partial implementation of polymorphism with some function and operator overloading.	No polymorphism implemented.	10	
STL	Excellent STL implementation with self-developed template libraries.	Adequate use of STL.	Minimal use of STL.	Partial or improper use of STL	No STL used.	10	
Files	Excellent implementation of file usage for data storage and retrieval. This includes searching and retrieving algorithm.	Adequate use of files for storage. Data and configuration information is also stored and retrieved from files.	Minimal use of files for storage. All required data are stored and retrieved from files.	Partial use of files for storage. Not all data are stored / retrieved from file	No file storage implemented.	5	

