VIETNAM NATIONAL UNIVERSITY - HCM
Ho Chi Minh City University of Technology
Faculty of Computer Science and Engineering

SOFTWARE ENGINEERING (CO3001)

Individual Submission #4

Smart Food Court System - SFCS

Instructor: Assoc. Prof. Quan Thanh Tho

Name	Student ID
Nguyễn Hữu Trung Nhân	**1752392**

Group Members	Student ID
Nhan Ngoc Thien	1752508
Tran Dinh Tien	1752541
Nguyen Viet Khoa	1752295
Pham Nhat Phuong	1752042
Nguyen Huu Trung Nhan	1752392

Ho Chi Minh City, June 2020 - Semester 192

Contents

1 Introduction to project **2**

2 Use case of the project **2**

3 Project features in general **3**

4 Non-functional features **4**

5 Non-interactive functional requirements **4**

6 "Make Payment" Feature **5**
6.1 Use-Case Diagram 5
6.2 Use-Case Specification 6

7 "Place Order" Feature **7**
7.1 Use Case Diagram 7
7.2 Use Case Specification 8

8 Activity Diagrams **9**
8.1 Activity Diagram for "Make Payment" Feature 9
8.2 Activity Diagram for "Place Order" Feature 10

9 Sequence Diagrams **11**
9.1 Sequence Diagram for "Make Payment" Feature 11
9.2 Sequence Diagram for "Place Order" Feature 12

10 State Diagrams **13**
10.1 State Diagram for "Make Payment" Feature 13
10.2 State Diagram for "Place Order" Feature 14

11 Deployment View **15**

12 Development View **16**
12.1 Component Diagram 16
12.2 Package Diagram 19

13 Module Interface **21**

14 Class Diagram **31**

15 Method Description **32**

16 Sequence Diagram At Detail Level **36**

17 Activity Diagram At Detail Level **37**

18 Design Pattern **38**

19 A working demonstration **39**

1 Introduction to project

With the conventional way of ordering food in the canteen or food court, the students must stand in a line and wait until their turn to get the food. This can take a very long time in rush hour such as lunch time and dinner. In order to improve the quality in comparison with the conventional way, the Smart Food Court System is designed to help the students to save more time with modern food court system. The system can take the orders of students directly at the vendor machines or through online services. Once the order is taken, the vendor machines will give the students the ordinal number and students can wait at the waiting are until the food is ready. For the online service, students are also get a notification when the food is ready to serve. With the Smart Food Court System, the process of ordering food can be more efficient and time-saving.

2 Use case of the project

(Please zoom in to see more clearly)

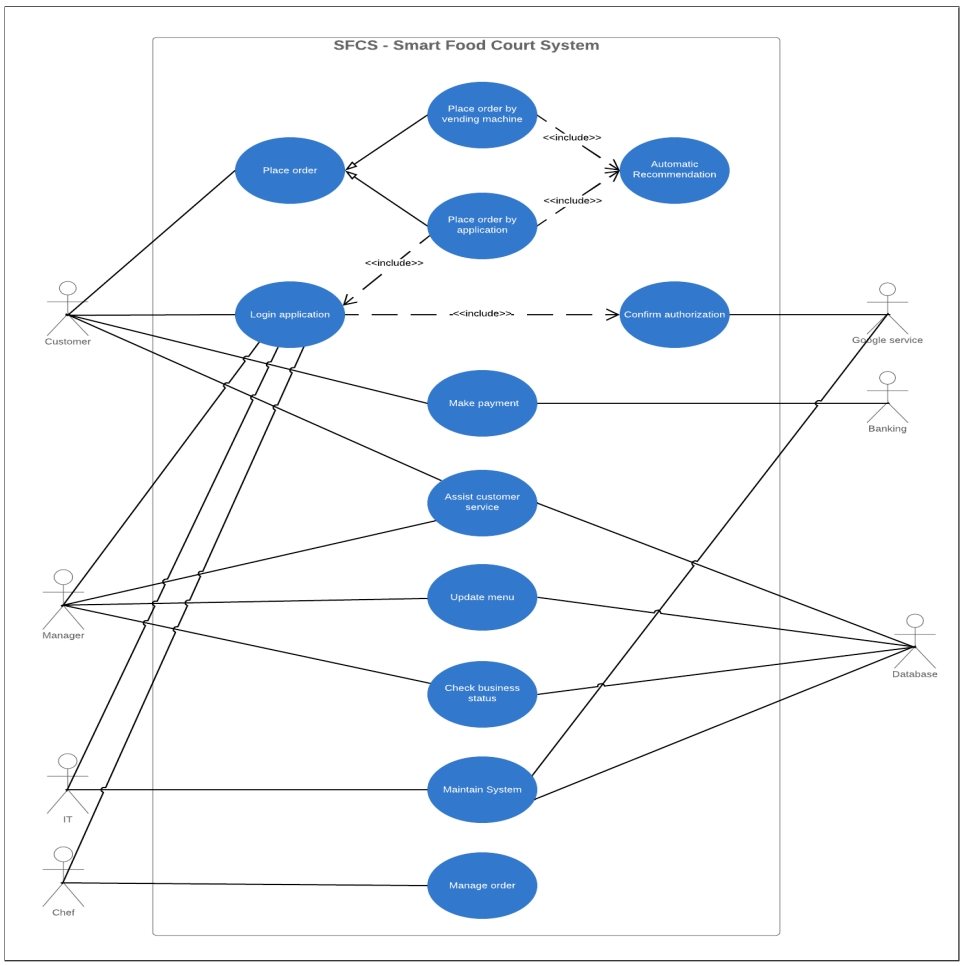

Figure 1: Use-case diagram for the whole system

3 Project features in general

Feature 1: Provide the ordinal number in the order list for the customer
As soon as a customer submits an order (ordering directly at the food court or using the order app), the smart food court system (SFCS) will verify the order and sent it to the chef.

Feature 2: The system provides a friendly User Interface for the customers
The customers can view the price of food, drinks and select the quantities they want to order. The system will notify if the amount of food the customers order is not enough to cook.

Feature 3: The system provides a pager that notifies the customers when the food is ready.
The remaining time will be display in minutes on the pager. When the time is up, the pager rings to alert the customers to come and collect the ordered food.

Feature 4: The payment can be made by some online payment service.
If the customers have cash in the online banking service, the customers can pay via online payment service such as MoMo wallet, Samsung Pay, Apple Pay, etc

Feature 5: The system has a mobile app for members to order food before comes to the food court.
For the members of the system, they can order the food before coming to the food court. All the information will be notified to the customers via the app as well as the remaining time until the food is ready.

Feature 6: The system has an food and drinks automatic recommendation function.
Whenever customers have ordered food, drinks and the order has not been confirmed, the system will recommend popular food, drinks based on how many times they have been ordered and are displayed through a percentage chart.

Feature 7: The system has an customer care services for the customer.
After finishing the meal, customers can comment and leave some reviews or complaints on the application. After that they will be delivered to the manager in order to improve the system quality. Moreover, customers can directly contact the manager for faster response.

4 Non-functional features

Feature 1: The delay of data transmission should be less than 0.01s.

Feature 2: The User Interface of the application is friendly and easy to use.

Feature 3: The smart food court system (SFCS) should be operational 24/7 without fail.

Feature 4: The size of the application must be no larger than 10 MB.

Feature 5: The application is cross-platform (The application should be supported on both Android and iOS).

Feature 6: The smart food court system (SFCS) should be capable enough to handle 1,000,000 users with affecting its performance.

Feature 7: Privacy of information of the customer (such as Identification Card Number, mobile phone number, bank account number,...) as well as intellectual property rights should be audited.

Feature 8: The mobile application should be portable in order to avoid problems when moving from one OS to other OS.

5 Non-interactive functional requirements

Feature 1: Vending Machine changes to power-saved mode if it is not used by anyone within 5 minutes.

Feature 2: The machine automatically shuts down at 19:00 and automatically turns on at 05:00 everyday.

Feature 3: The system automatically updates the profit status and saves this document in the database.

Feature 4: Recommendation System uses filtering algorithm to recommend favorite dishes or food combos to the customers.

6 "Make Payment" Feature

6.1 Use-Case Diagram

(Please zoom in to see more clearly)

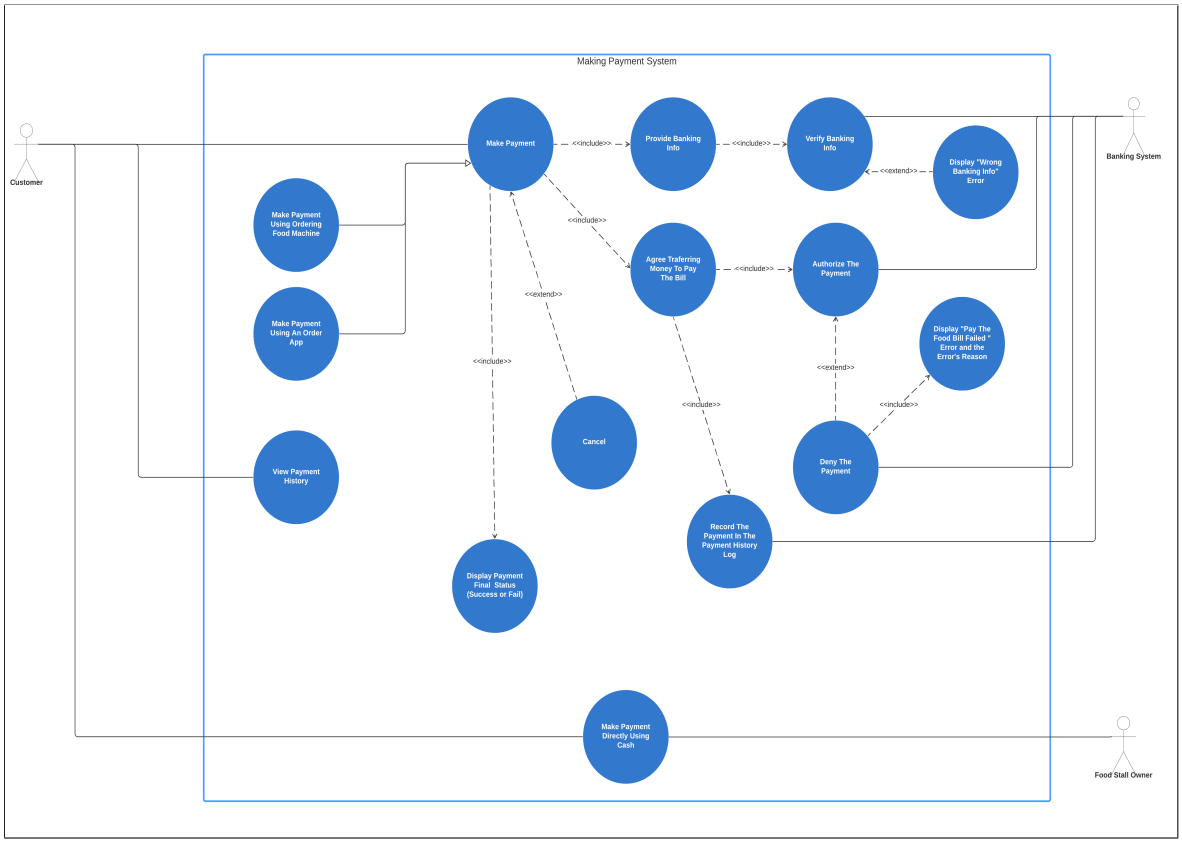

Figure 2: Use-case diagram for "Make Payment" Feature

6.2 Use-Case Specification

Use-case Name:	Make Payment
Primary Actor(s):	Customer
Secondary Actor(s):	Banking System, Food Stall Owner
Description:	The customer uses this "Make Payment" feature to pay the food bill
Preconditions:	The customer has submitted the food order successfully. The customer has logged in successfully if he/she wants to pay the bill using online banking system.
Normal Flow:	1. The user choose a method to make payment. 2. The user provide his/her banking information (traditional bank, credit card, Mo Mo wallet, Samsung Pay,...). 3. The Making Payment System will have this banking information verified by the Banking System. 4. The customer confirms to transfer the money to pay the food bill. 5. The Banking System authorizes the payment. 6. The Banking System records the payment in the payment history log. 7. The system displays a message to inform the making payment process successfully completed.
Exceptions:	**Exception 1:** at step 2 2a. The customer cancels "Make Payment" process. **Exception 2:** at step 4 4a. The customer cancels "Make Payment" process. **Exception 3:** at step 3 3a. The customer's banking information is incorrect. Back to step 2 in the Normal Flow. **Exception 4:** at step 5 5a. The Banking System informs that the amount of money in the bank account is not enough to pay the bill.
Alternative Flows:	**Alternative 1:** at step 1 1a. The customer uses Ordering Food Machine to make payment. Continue step 2 in the Normal Flow. **Alternative 2:** at step 1 1b. The customer uses the Order App to make payment. Continue step 2 in the Normal Flow. **Alternative 3:** at step 1 1c. The customer uses cash to pay the bill, the customer gives the money directly to the food stall owner.
Non-Functional Requirement(s):	+The payment processing gateway must be PCI DSS compliant. +The Making Payment System shall be available for use 24/7. +The Making Payment System should allow no one to get access to the customer's banking information.

7 "Place Order" Feature

7.1 Use Case Diagram

(Please zoom in to see more clearly)

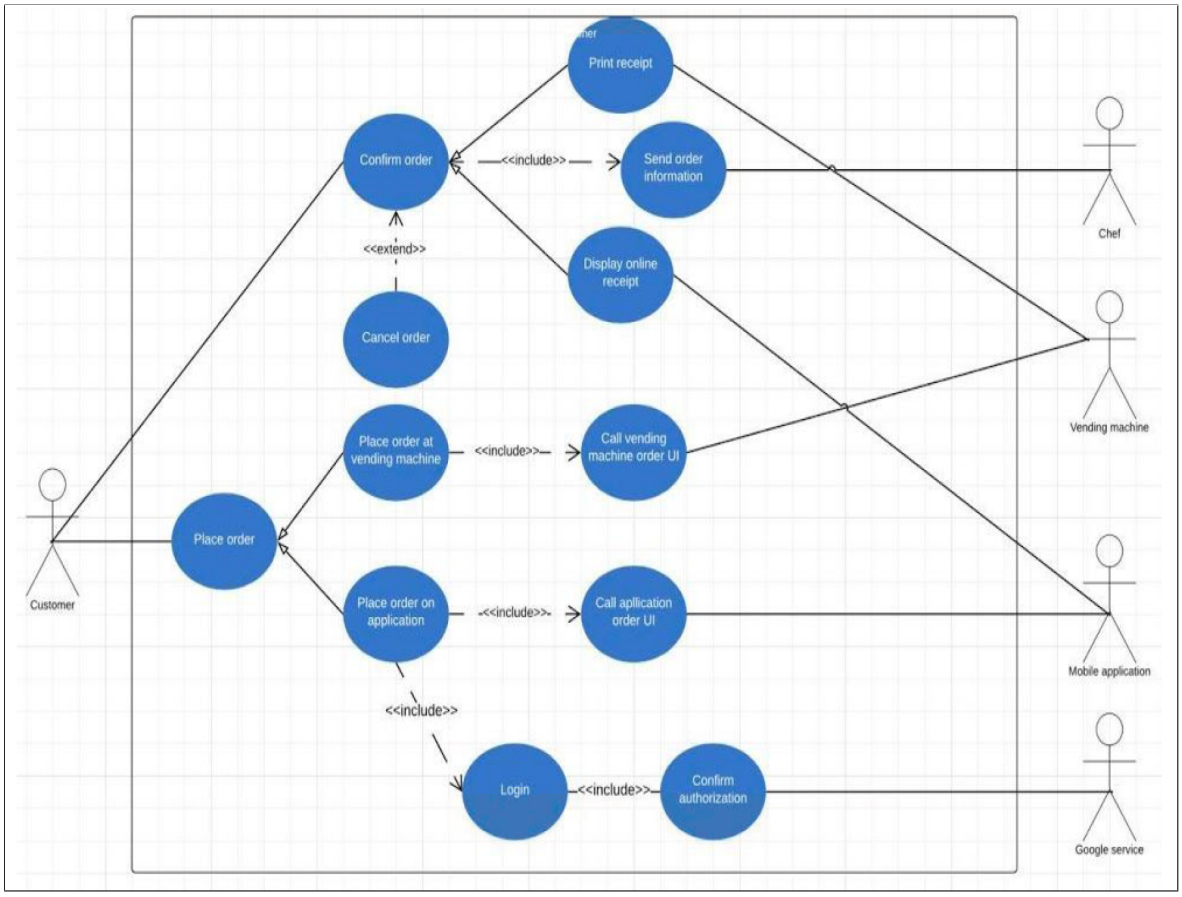

Figure 3: Use-case diagram for "Place Order" Feature

7.2 Use Case Specification

Use-case Name:	Place Order
Primary Actor(s):	Customer
Secondary Actor(s):	Mobile App, Vending Machine, Chef
Description:	This use case describes how the user creates a new order for the restaurant.
Preconditions:	The database is up and running. The customer has logged in successfully on mobile phone App if he/she wants to place order on the App.
Normal Flow:	**Case A:** User orders on the machine at food court 1. User chooses menu kind (food or beverage). 2. User chooses items in the menu screen. 3. User chooses quantity for each item. 4. User selects "Confirm" button in order to confirm that each selected item is corrected. 5. User is received a receipt from vending machine with ordered number. 6. System sends his/her order information to the "Check order" function and show prepared condition ("waiting", "cooking") to user. **Case B:** User orders on application. 1. User access "Login application" function. 2. User chooses menu kind (food or beverage). 3. User chooses items in the menu screen. 4. User chooses quantity for each item. 5. User can set the time to pick up pre-ordered food. 6. User selects "Confirm" button in order to confirm that each selected item is corrected. 7. System sends his/her order information to the "Check order" function and show prepared condition ("waiting", "cooking") to user.
Exceptions:	There is no exceptions for this use-case.
Alternative Flows:	**Alternative 1:** at case A step 4 A.4a. User selects "Cancel" button in order to cancel current order and reorder. **Alternative 2:** at case B step 6 B.6a. User selects "Cancel" button in order to cancel current order and reorder.
Non-Functional Requirement(s):	+The loading time of the menu should be less than 0.5 sec. +The drop down list in the menu is arrange alphabetically (From A to Z).

8 Activity Diagrams

8.1 Activity Diagram for "Make Payment" Feature

(Please zoom in to see more clearly)

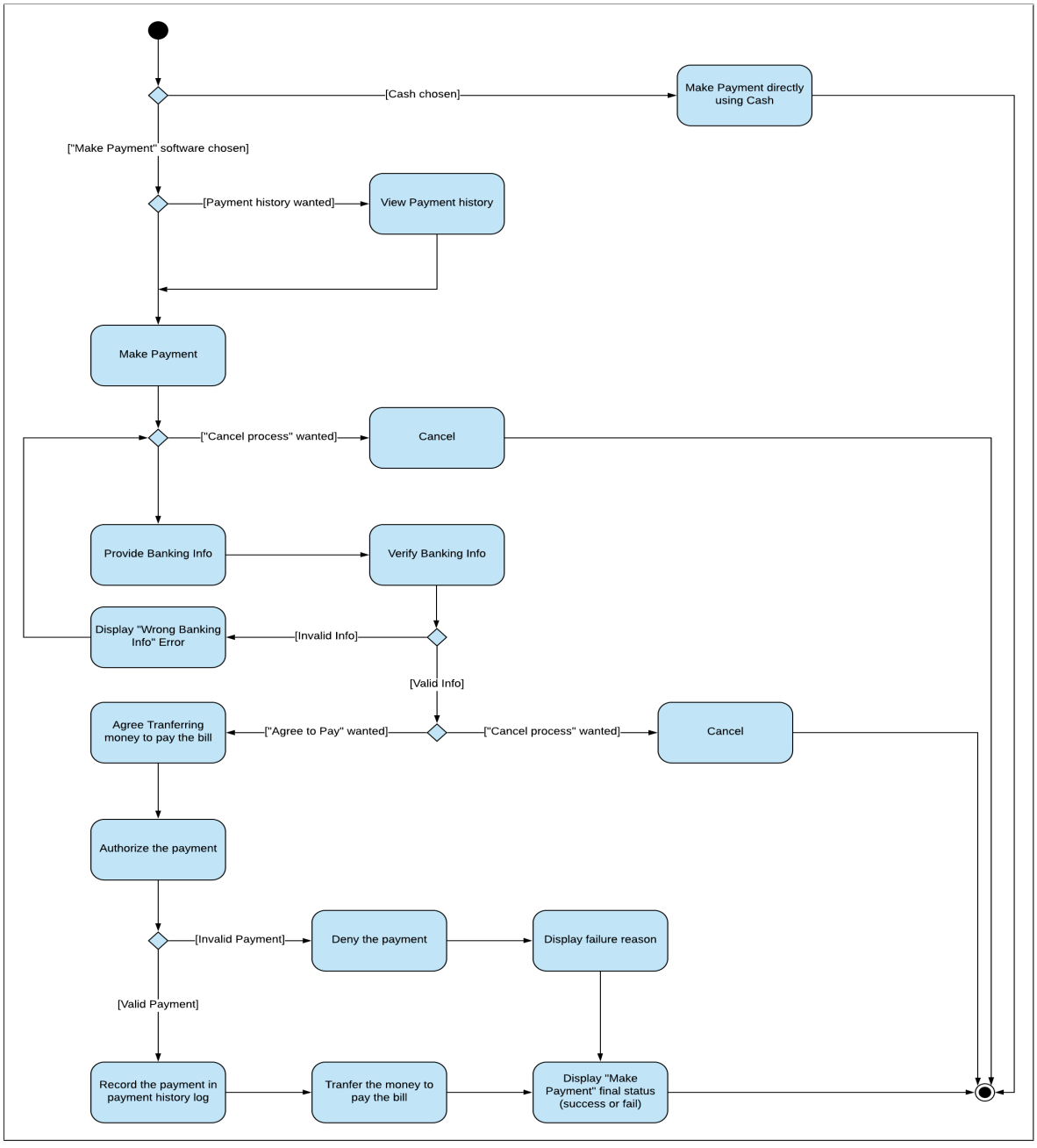

Figure 4: Activity diagram for "Make Payment" Feature

8.2 Activity Diagram for "Place Order" Feature

(Please zoom in to see more clearly)

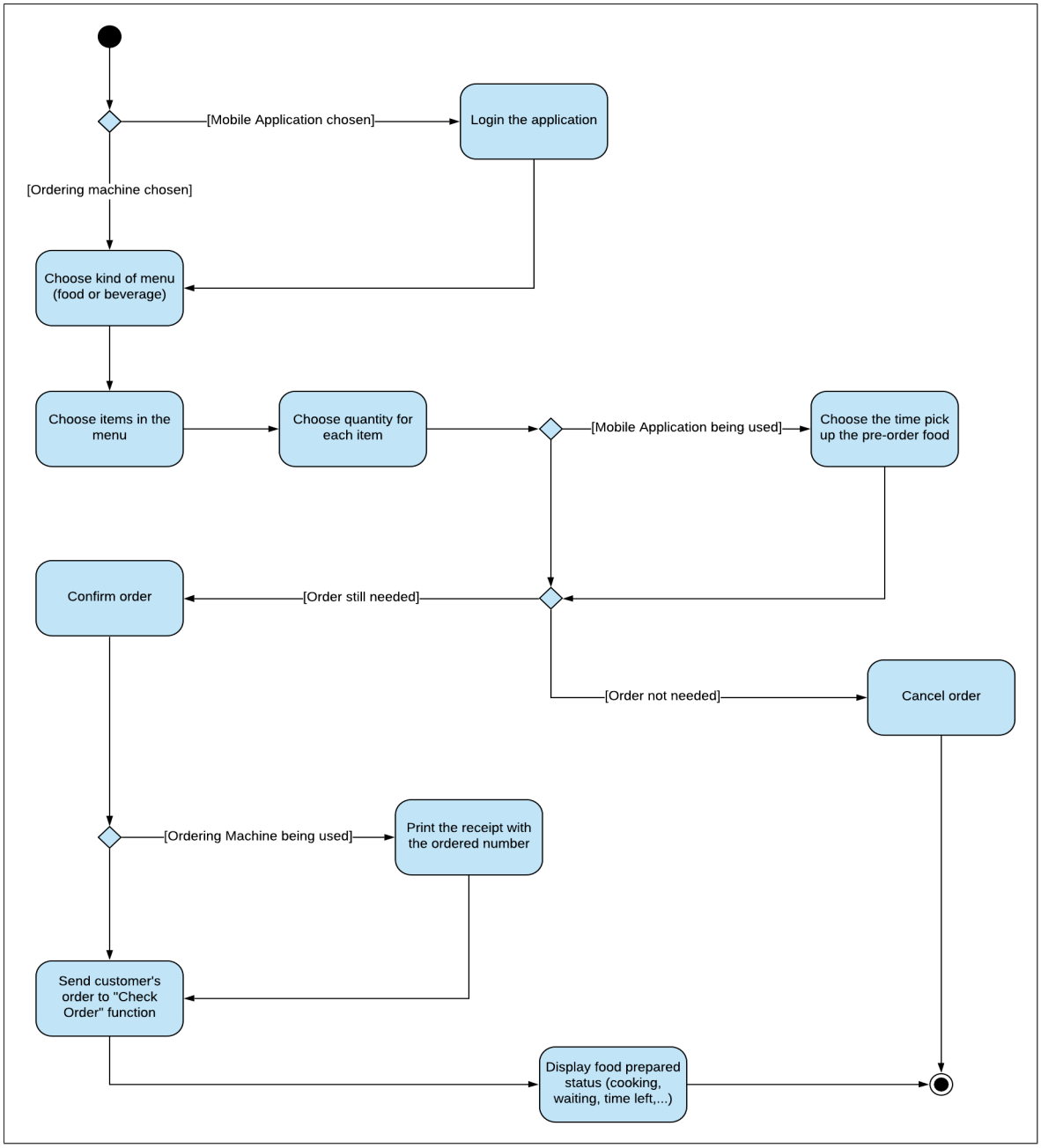

Figure 5: Activity diagram for "Place Order" Feature

9 Sequence Diagrams

9.1 Sequence Diagram for "Make Payment" Feature

(Please zoom in to see more clearly)

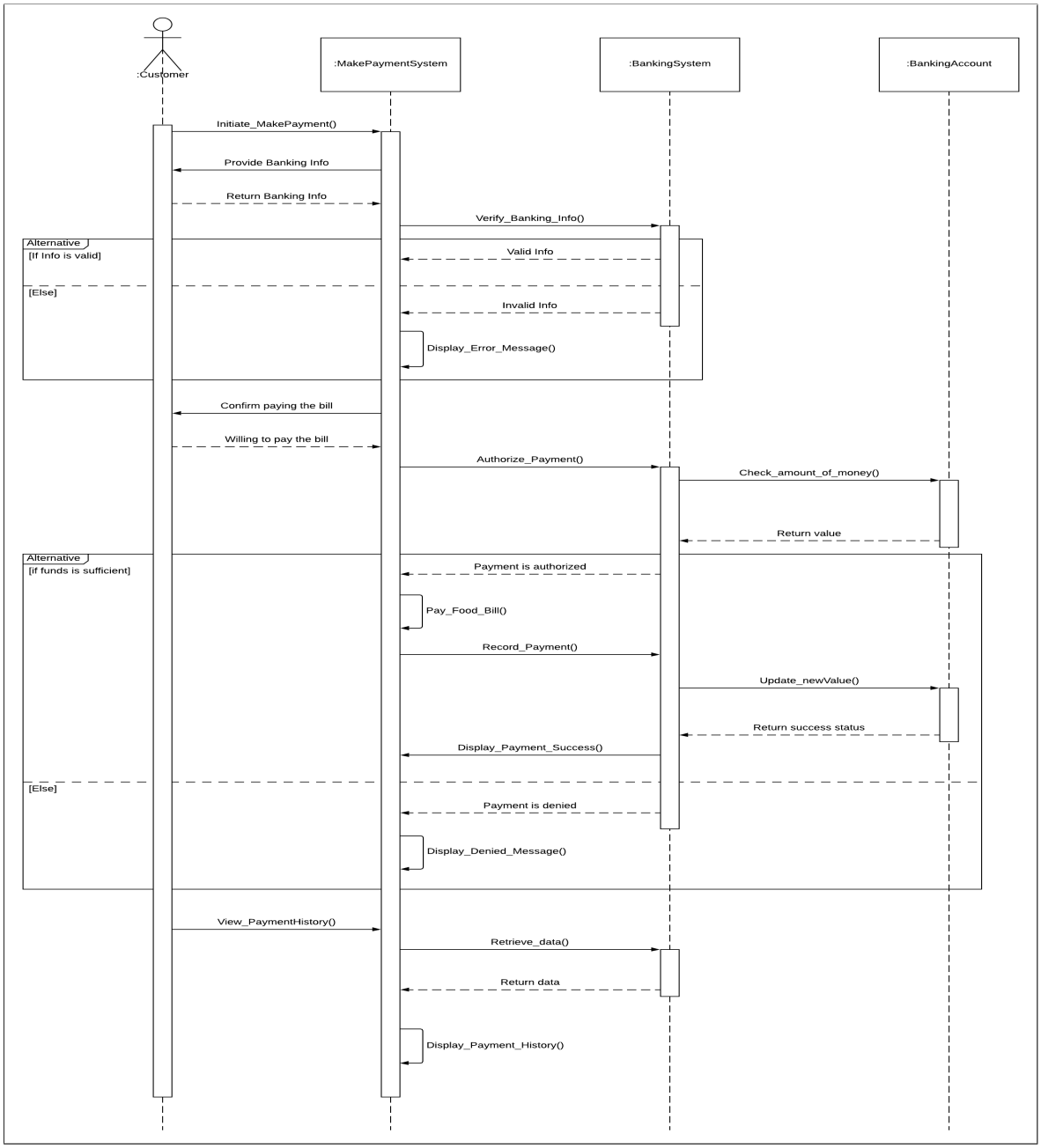

Figure 6: Sequence diagram for "Make Payment" Feature

9.2 Sequence Diagram for "Place Order" Feature

(Please zoom in to see more clearly)

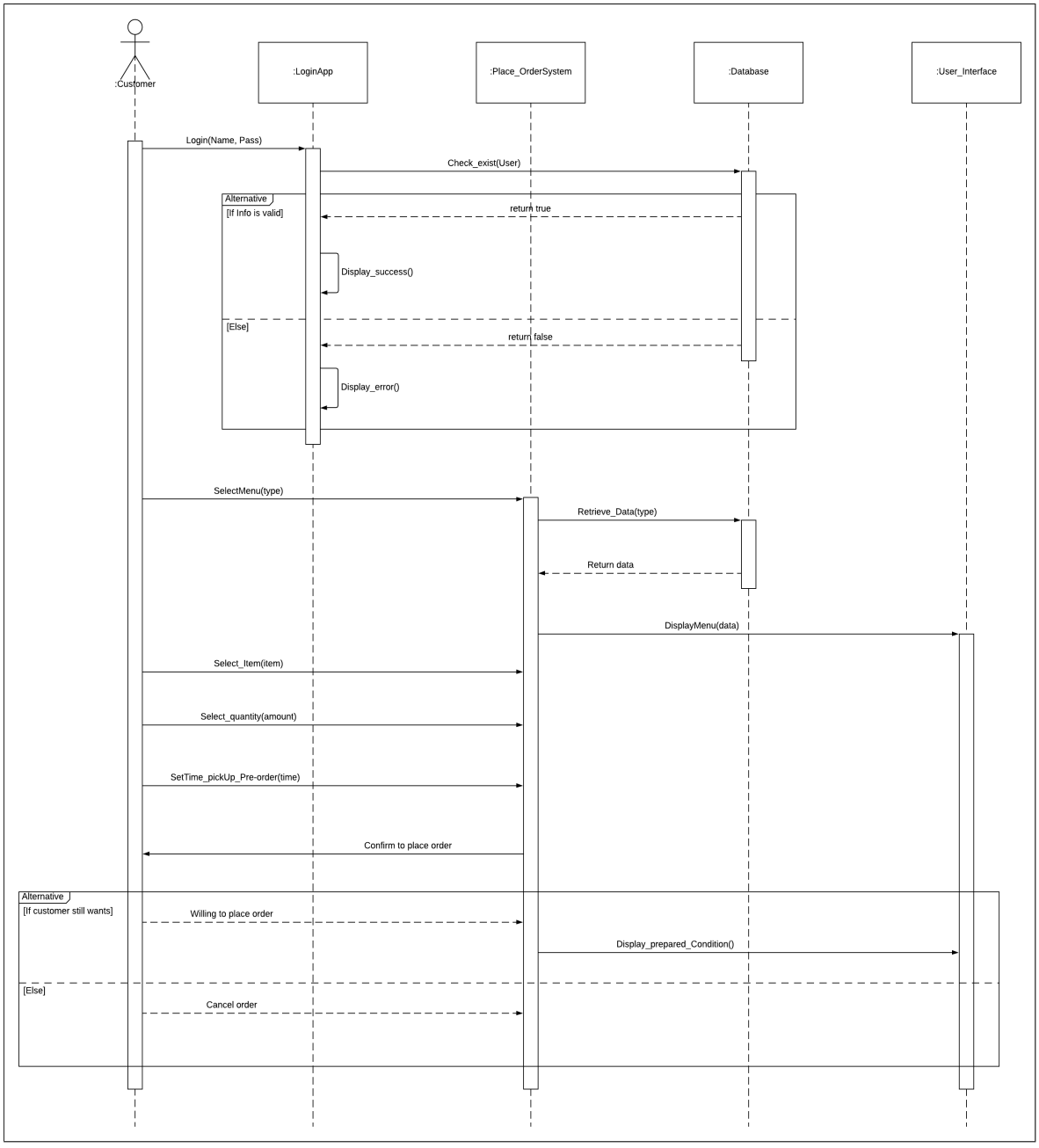

Figure 7: Sequence diagram for "Place Order" Feature

10 State Diagrams

10.1 State Diagram for "Make Payment" Feature

(Please zoom in to see more clearly)

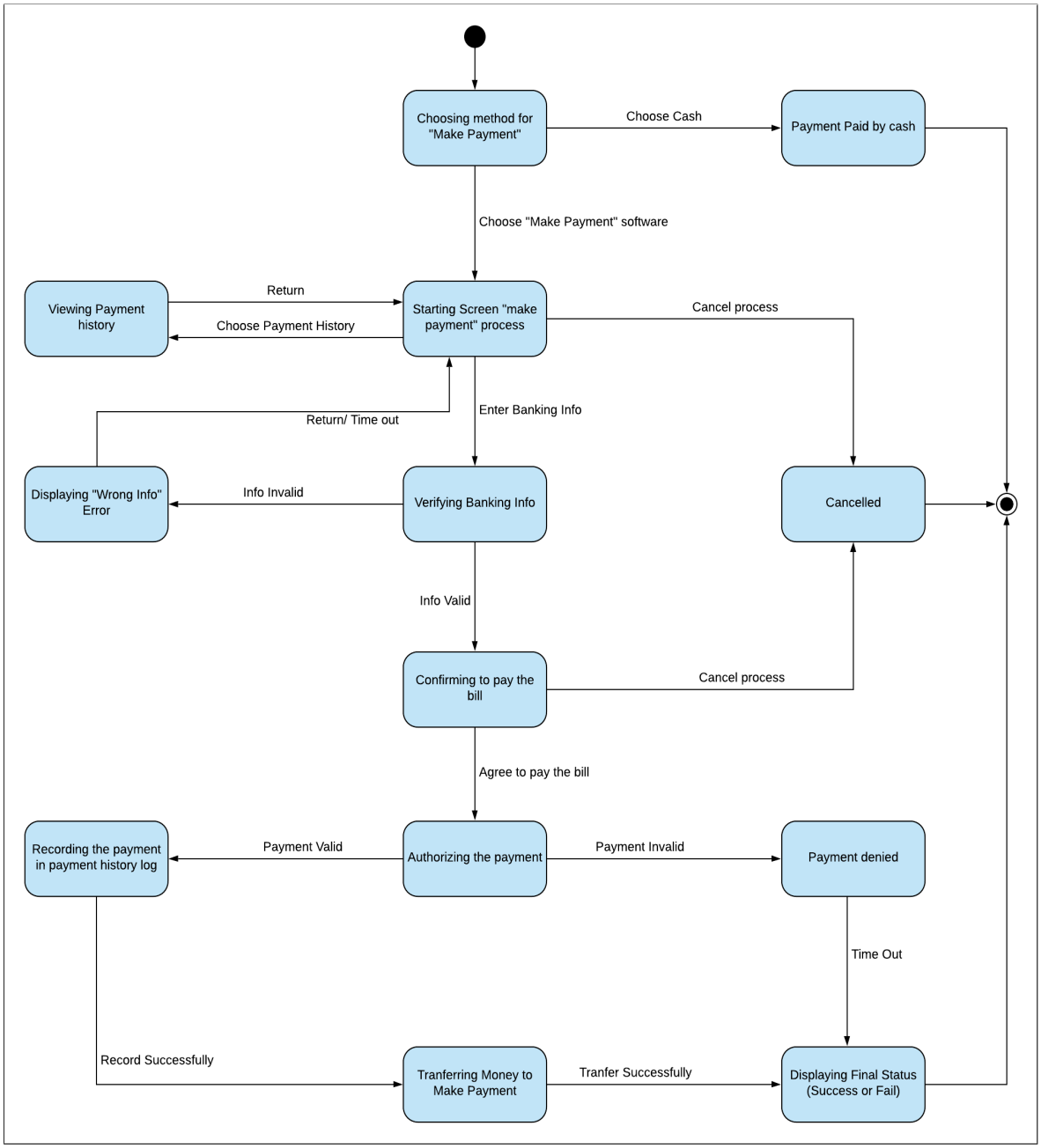

Figure 8: State diagram for "Make Payment" Feature

10.2 State Diagram for "Place Order" Feature

(Please zoom in to see more clearly)

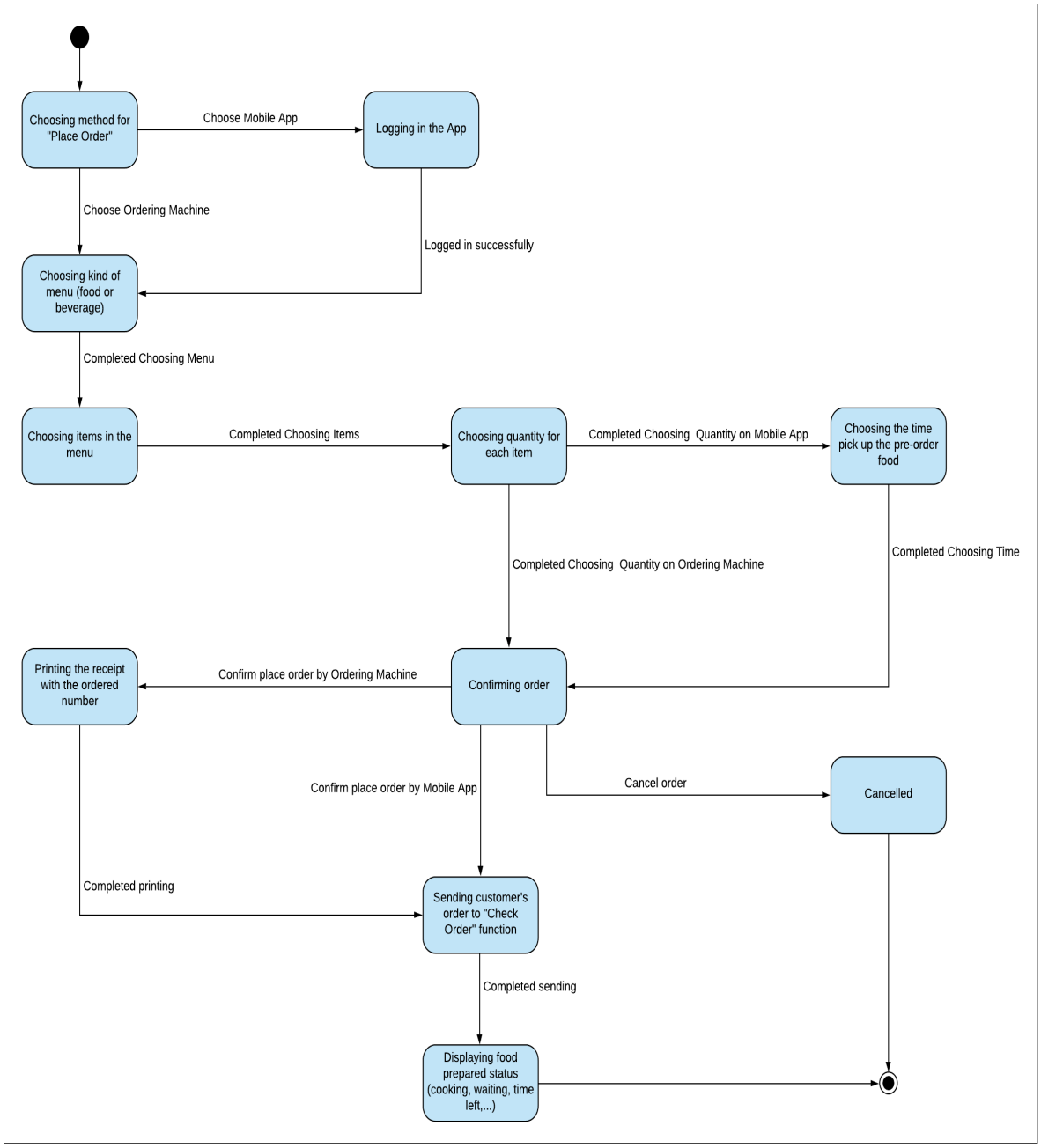

Figure 9: State diagram for "Place Order" Feature

11 Deployment View

(Please zoom in to see more clearly)

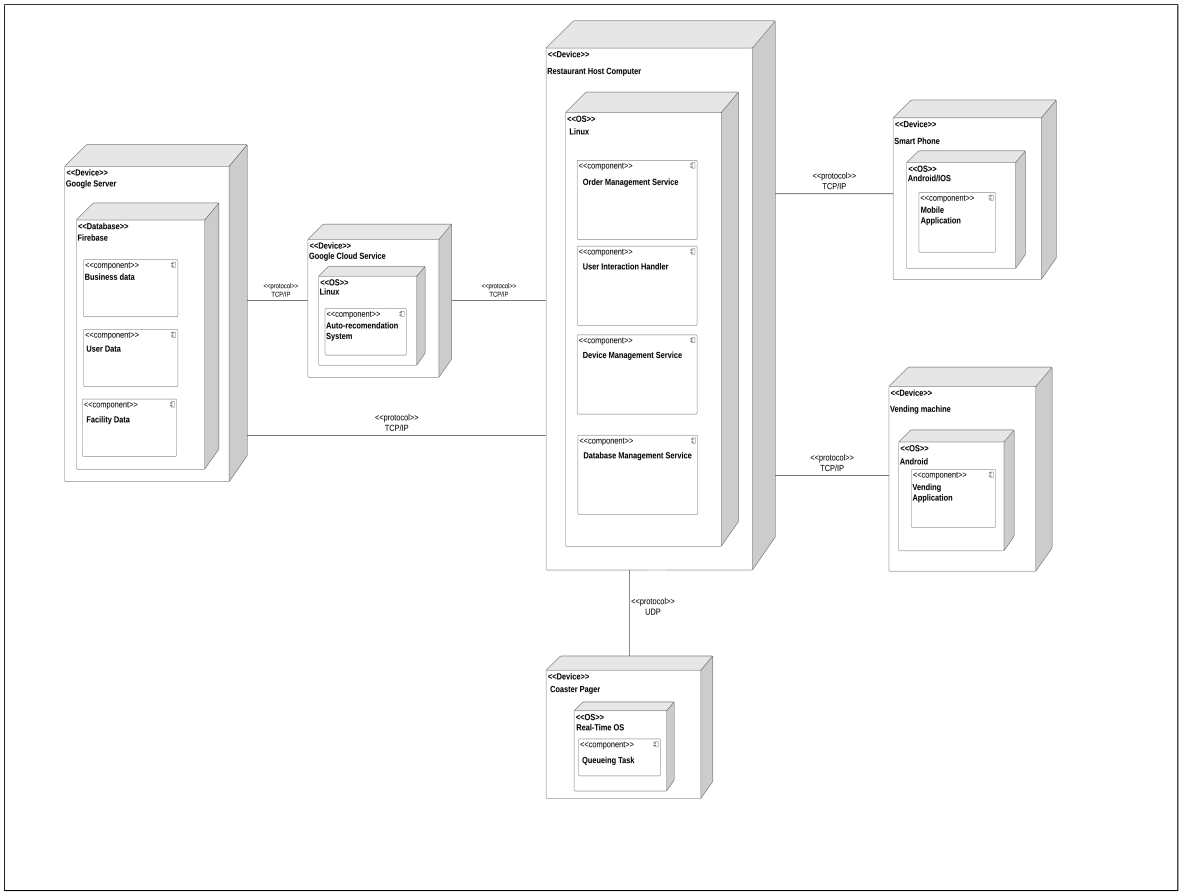

Figure 10

Work-space link of this diagram:
https://app.lucidchart.com/invitations/accept/eaec5cc8-86dc-4b10-889f-508d02742334

12 Development View

12.1 Component Diagram

(Please zoom in to see more clearly)

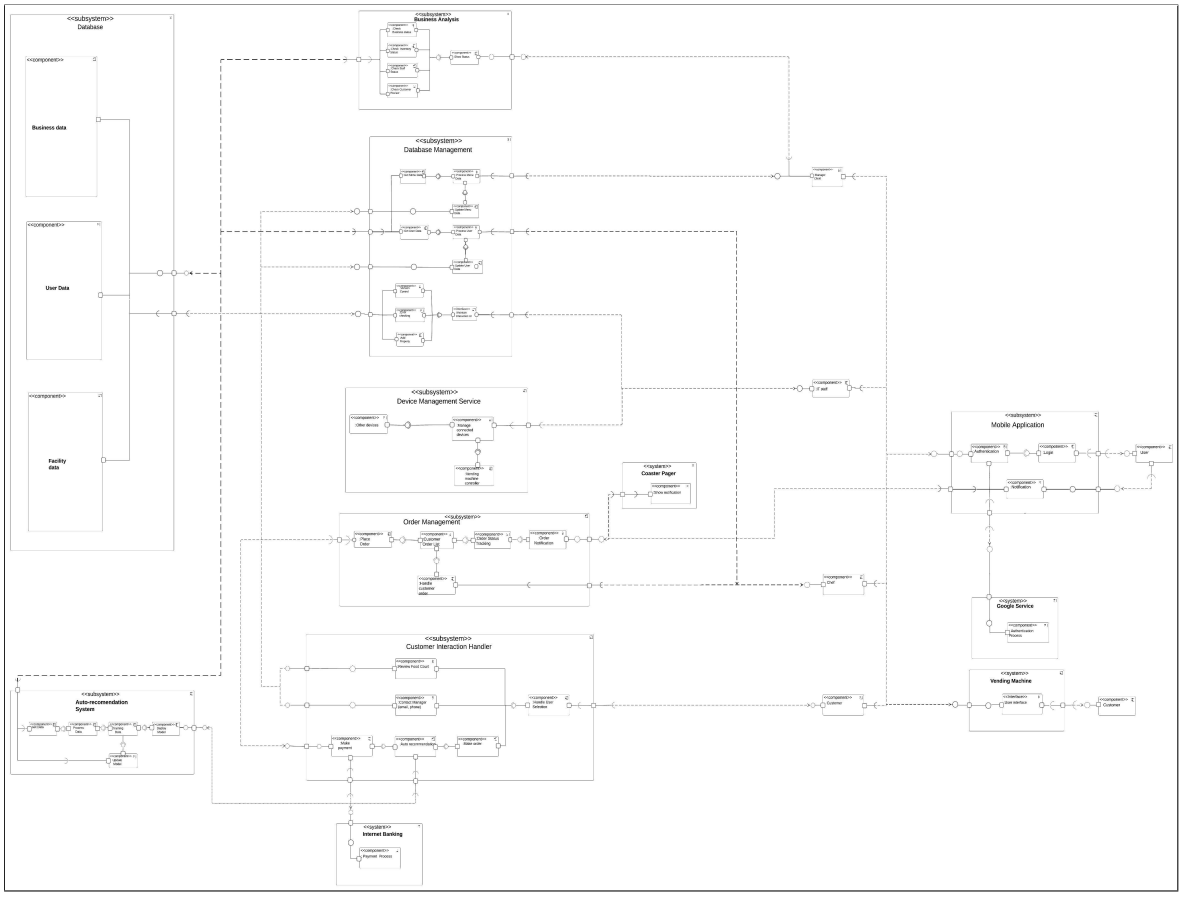

Figure 11

Work-space link of this diagram:
https://app.lucidchart.com/invitations/accept/24bd22fa-baff-4e2b-aa3a-a8da109ce87f

(Please zoom in to see more clearly)

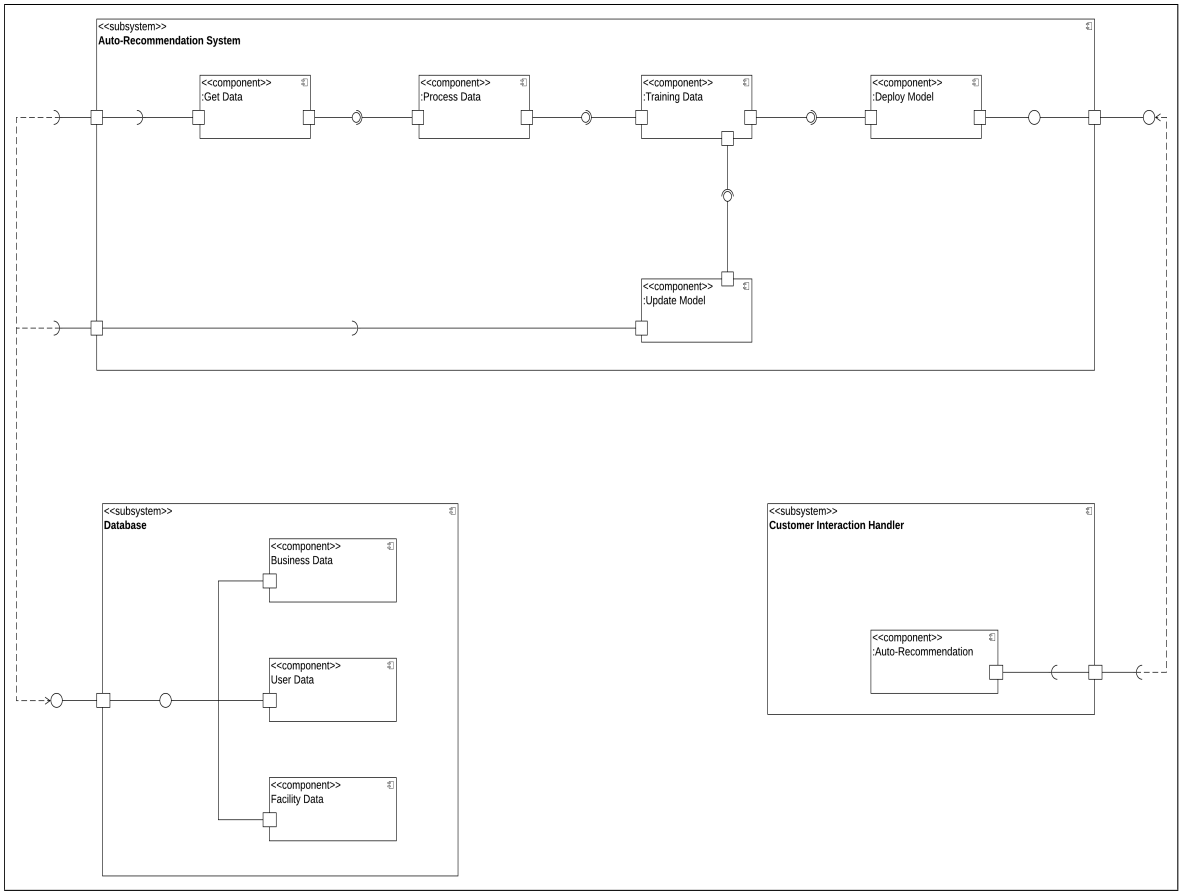

Figure 12: Component diagram of "Auto-Recommendation System"

(Please zoom in to see more clearly)

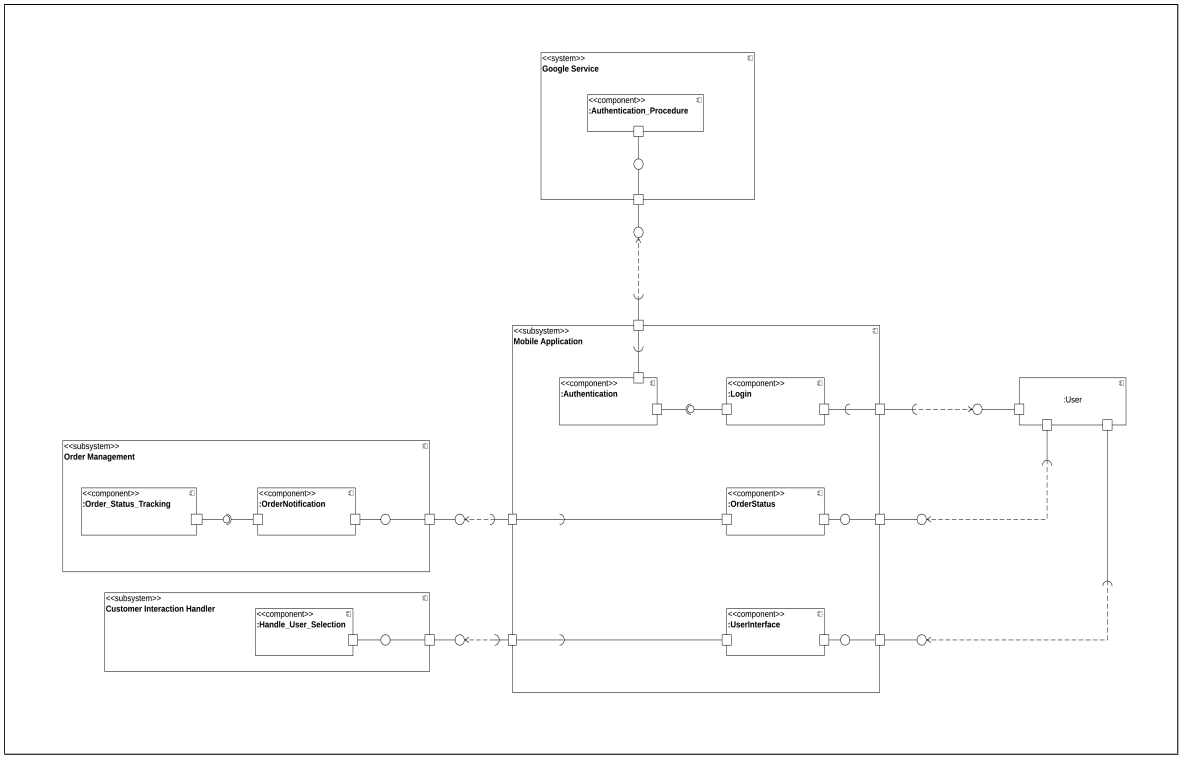

Figure 13: Component diagram of "Mobile Application"

12.2 Package Diagram

(Please zoom in to see more clearly)

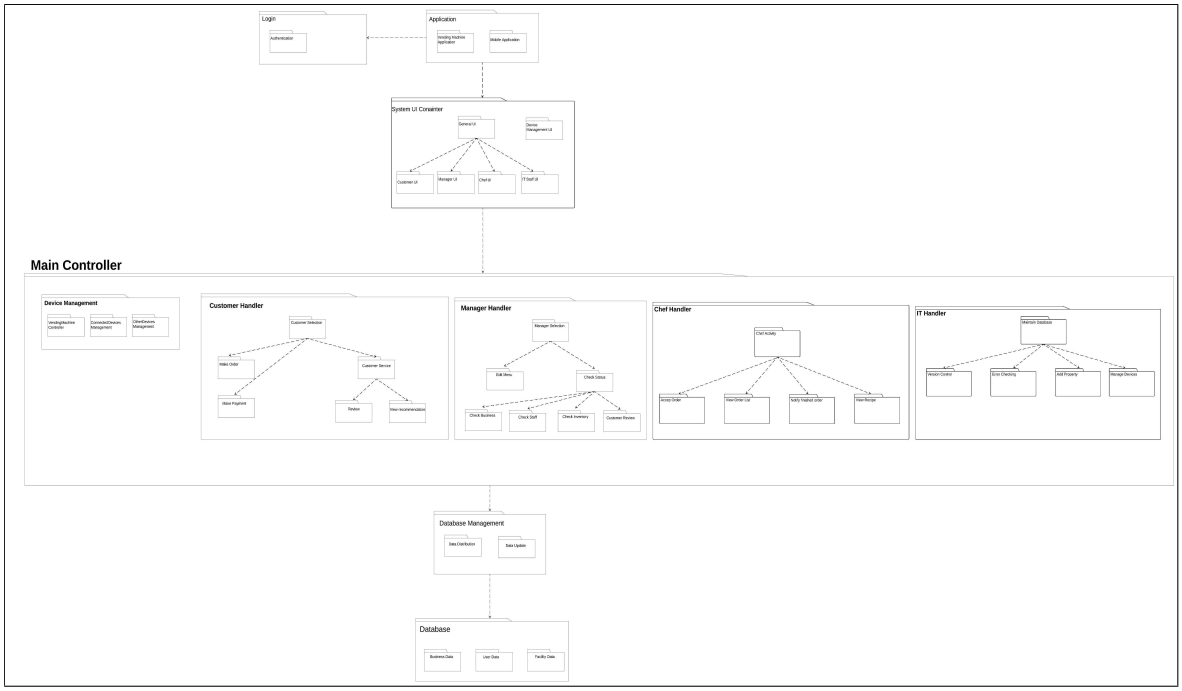

Figure 14

Work-space link of this diagram:
https://app.lucidchart.com/invitations/accept/1633c3b2-6040-4e33-b2f7-87637ea4c89c

(Please zoom in to see more clearly)

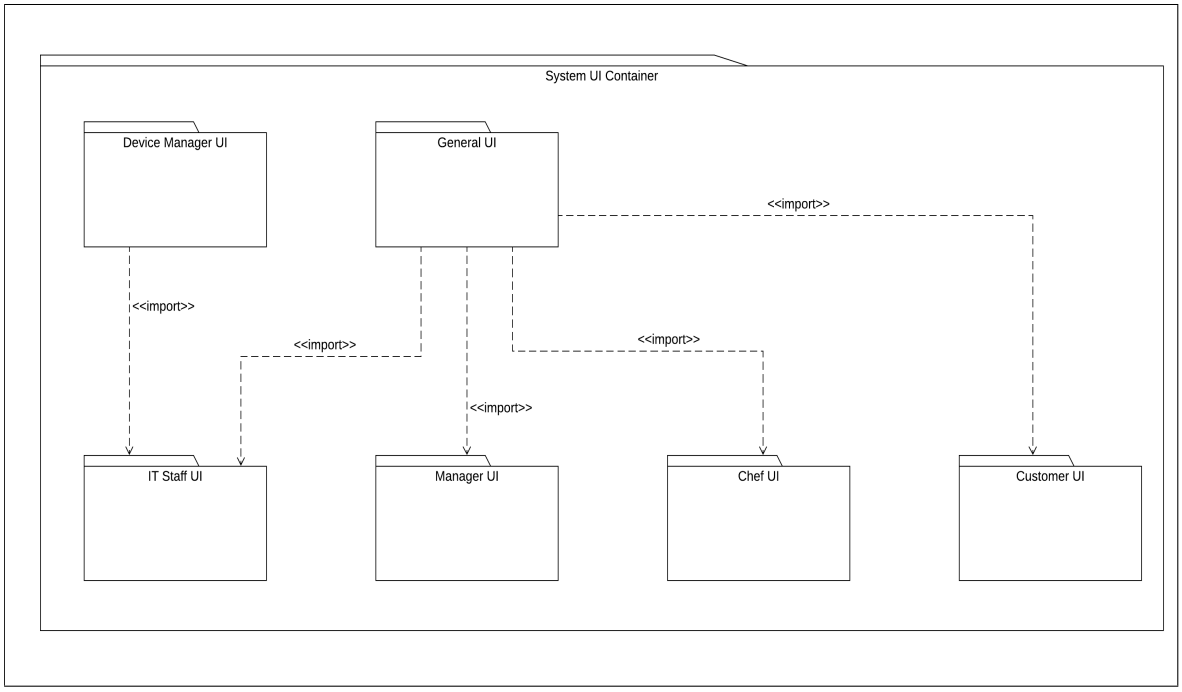

Figure 15: Package diagram of "System UI Container"

13 Module Interface

System UI

```
class LoginForm{
   public:
      void onClickLogin(){
         return;
      }
      Integer getToken(String userName, String userPassword){
         return Integer feedbackToken;
      }
      Boolean showLoginStatus(Integer feedbackToken){
         return boolean loginStatus;
     }
}
```

```
class ChefUI{
   public:
      void create() {
         return;
      }
      void displayChefUI() {
         return;
      }
      void eventListener(String event){
         return;
      }
 }
```

```
class VendingMachineUI{
   public:
      void displayVendingUI(){
         return;
      }
      void eventListener(String event){
         return;
      }
      void Create() {
          return;
      }
}
```

```
class ITStaffUI{
   public:
      void create(){
         return;
      }
      void displayITStaffUI(){
         return;
      }
      void eventListener(String event){
         return;
```

```
    }
}
```

```
class CustomerUI{
   public:
      void create(){
         return;
      }
      void displayManagerUI(){
         return;
      }
      void eventListener(String event){
         return;
      }
}
```

```
class ManagerUI{
   public:
      void create() {
         return;
      }
      void displayManagerUI(){
         return;
      }
      void eventListener(String event){
         return;
      }
}
```

Controller

```
class ChefController{
   private:
      List<FoodItem> checkOrderList(Order a){
         return List<FoodItem> listOrder;
      }
      Order chooseOrder(int orderID){
         return Order chosenOrder;
      }
      Boolean sendNotify(Order a){
         if (orderFinished()){
             return true;
           }
         return false;
      }
}
```

```
class CustomerController {
private:
  Order dishList;
  Order makeOrder() {
    return dishList;
  }
```

```
  void chooseDish(Order dishList, Dish dish) {
      return;
   }
  void makePayment() {
      return;
   }
  void checkOrderStatus(Order order) {
      return;
   }
  void contactManager(User manager) {
      return;
   }
  void reviewFood(User customer, Order order) {
      return;
   }
}
```

```
class ITStaffController{

   private:
      void setflag(boolean status){
         return flag;
      }
      List<deviceType> getListByType(String deviceType){
         return List<deviceType>
      }
      void searchDevice(String deviceType,  String Device ID){
         return deviceType deviceInfo;
      }
      void addDevice(String deviceType, String ID, String description,
          String state){
         return;
      }
      void removeDevice(String deviceType, String ID){
         return;
      }
      void modifyDevice(String deviceType, String Property){
         return;
      }
      void setDeviceState(DeviceList deviceList){
         return;
      }
      String versionControl(){
         return version;
      }
      void train() {
         return;
      }
      void updateSystem(){
         return;
      }
      void checkErrors(){
         return;
```

```
    }
    void fixErrors(){
        return;
    }
    void turnOnServices(){
        return;
    }
}
```

```
class ManagerController{
   private:
      void showBusinessStatus(List<Payment> paymentList){
         return;
      }
      void showInventorryStatus(List<FoodItem> foodList){
         return;
      }
      void modifyInventory(String modifyType, String modifyProperty){
         return;
      }
      void showStaffStatus(List<User> userList) {
         return;
      }
      void modifyStaff(String modifyType, String modifyProperty){
         return;
      }
      void modifyMenuList(String modifyType, String modifyProperty){
         return;
      }
      void showReview(Review customerReview) {
         return;
      }
      void showContact() {
         return;
      }
      void deleteUser() {
         return;
      }
}
```

User Service

```
class RecommendationSystem {
private:
  Integer trainingPeriod;
  Integer epoch;
  Integer batchSize;
  List<Data> processData(Order order) {
    return List<Data> dataList;
  }
  void trainModel(List<Data> dataList, &model) {
      return;
  }
public:
```

```
  List<Dish> getRecommendedList(Order order, model) {
    return List<Dish> foodItem;
  }
   void setParameters(Integer trainingPeriod, Integer epoch, Integer
       batchSize){
      return;
   }
   void sortByType(){
      return;
   }
}
```

System Database

```
class DatabaseFoodCourt{
   private:
      String version;
      List<User> userList;
      List<Dish> menuList;
      List<FoodItem> inventoryList;
      List<Payment> paymentList;
      List<Review> reviewList;
      List<VendingMachine> vendingMachineList;
      List<CoasterPager> coasterPagerList;
      void changePermission(Integer userID, String permission){
         return;
      }
      void addUser(User newUser){
         return;
      }
      void removeUser(Interger userID){
         return;
      }
      void updateUser(String modifyType, String modifyProperty){
         return;
      }
   public:
      List<VendingMachine> getVendingList(){
         return vendingMachineList;
      }
      List<PagerList>  getPagerList(){
         return coasterPagerList;
      }
      void saveDeviceList(List<VendingMachine>  vendingList, List<
          PagerList> pagerList)  {
         return;
      }
      List<Dish> getMenuList(){
         return menuList;
      }
      List<FoodItem> getInventoryList(){
         return inventoryList;
      }
      List<Payment> getPaymentList(){
```

```
        return paymentList;
    }
    void storeReview(Review customerReview){
        return;
    }
    List<User> getStaffList(){
        return  staffList;
    }
    List<Payment> getPaymentList(){
        return paymentList;
    }
    void addMenuItem(List<Dish> menuList){
        return;
    }
}
```

```
class CoasterPager {
private:
  Integer pagerID;
  Boolean isActive;
   String description;
public:
  Integer getPagerID() {
    return pagerID;
  }
  void setPagerID() {
  }
  Boolean getPagerStatus() {
    return isActive;
  }
  void setPagerStatus() {
      return;
  }
  void getNotify(Integer pagerID) {
      return;
  }
   void create() {
      return;
   }
   void addDescription(Integer pagerID) {
      return;
   }
   void delete(Integer pagerID) {
      return;
   }
}
```

```
class VendingMachine{
   private:
      Integer vendingID;
      Boolean isActive;
      String description;
      List<Order>  makeOrder(){
```

```
        return List<Order> customerOrder;
      }
      Payment makeCashPayment(){
        return Payment customerPayment;
      }
   public:
    void create(){
        return;
      }
      void delete(){
        return;
      }
      void setVendingStatus(){
        return;
      }
      void setVendingID(){
        return;
      }
      void addDescription(String userDescription) {
        return;
      }
   }
```

```
class DeviceList {

private:
  List<VendingMachine> vendingMachineList;
  List<CoasterPager> coasterPagerList;
  void modifyDevice(List<VendingMachine> vendingMachineList, List<
      CoasterPager> coasterPagerList) {
  }
public:
    void create(){
      return;
    }
  List<VendingMachine> getVendingList(){
    return List<VendingMachine> vendingMachineList;
  }
  List<CoasterPager> getPagerList() {
    return List<CoasterPager> coasterPagerList;
  }
   deviceType getInfoById(String deviceType, String deviceID){
      return deviceType deviceInfo;
   }
   void addToList(String deviceType, String ID, String state, String
       description){
      return;
   }
   void removeFromList(String deviceType, String ID){
      return;
   }
   void modifyDevice(String deviceType, String modifyType, String
       modifyProperty){
```

```
      return;
   }
   void  setDeviceStatus(String deviceType, String ID, boolean status){
      return;
   }
}
```

Other related module interfaces

```
class User{
   private:
      String userName;
      String userType;
      Integer userID;
   public:
      void setUserName{
            return;
      }
      void getUserName{
         return;
      }
      void setUserName{
         return;
      }
void getUserName{
            return;
      }
}
```

```
class Order {
   private:
      Integer orderID;
      Integer pagerID;
      String customerName;
      List<Dish> foodOrder;
      Boolean isFinished;
      List<Dish> getListOrderedFood() {
      return List<Dish> listOrderedFood;
   }
      Boolean checkFinishedStatus() {
         return True;
      }
   public:
      void addDish(DIsh dish){
         return;
      }
}
```

```
class FoodItem {
  private:
  String foodName;
  Integer foodID;
  String addedDate;
  String description;
  Boolean isAvailable;
  Integer quantity;
public:
  FoodItem getFoodInfo(){
    return FoodItem foodItem;
  }
}
```

```
class Recipe{
   public:
      List<FoodItem> ingredients;
      List<String> step;
      Integer estimatedTime;
      void modifyRecipe(String modifyType, Recipe recipe){
         return;
      }
}
```

```
class Review{
   private:
      String reviewerName;
      String content
      String reviewDate;
   public:
      void createReview(String reviewerName, String content, String
          reviewDate){
            return;
      }
}
```

```
class Dish{
   private:
   String dishName;
   Float dishPrice;
   String dishCategory;
   Recipe recipe;
   public:
      Dish getDishInfo() {
         return Dish dishInfo;
      }
}
```

```
class Payment {
private:
  String purchaseDate:
  Float purchaseAmount;
  String purchaseType;
  Boolean getBankAuthentication(String purchaseDate, Float purchaseAmount
      ) {
    return true;
  }
  boolean calculateCash(String purchaseDate, Float purchaseAmount) {
    return true;
  }
}
```

14 Class Diagram

Please zoom in to see more clearly, here is the link to the picture:
https://drive.google.com/drive/folders/1zTedMcN-OQBwm2iwPstmnBOa6aO7RBFh?usp=sharing

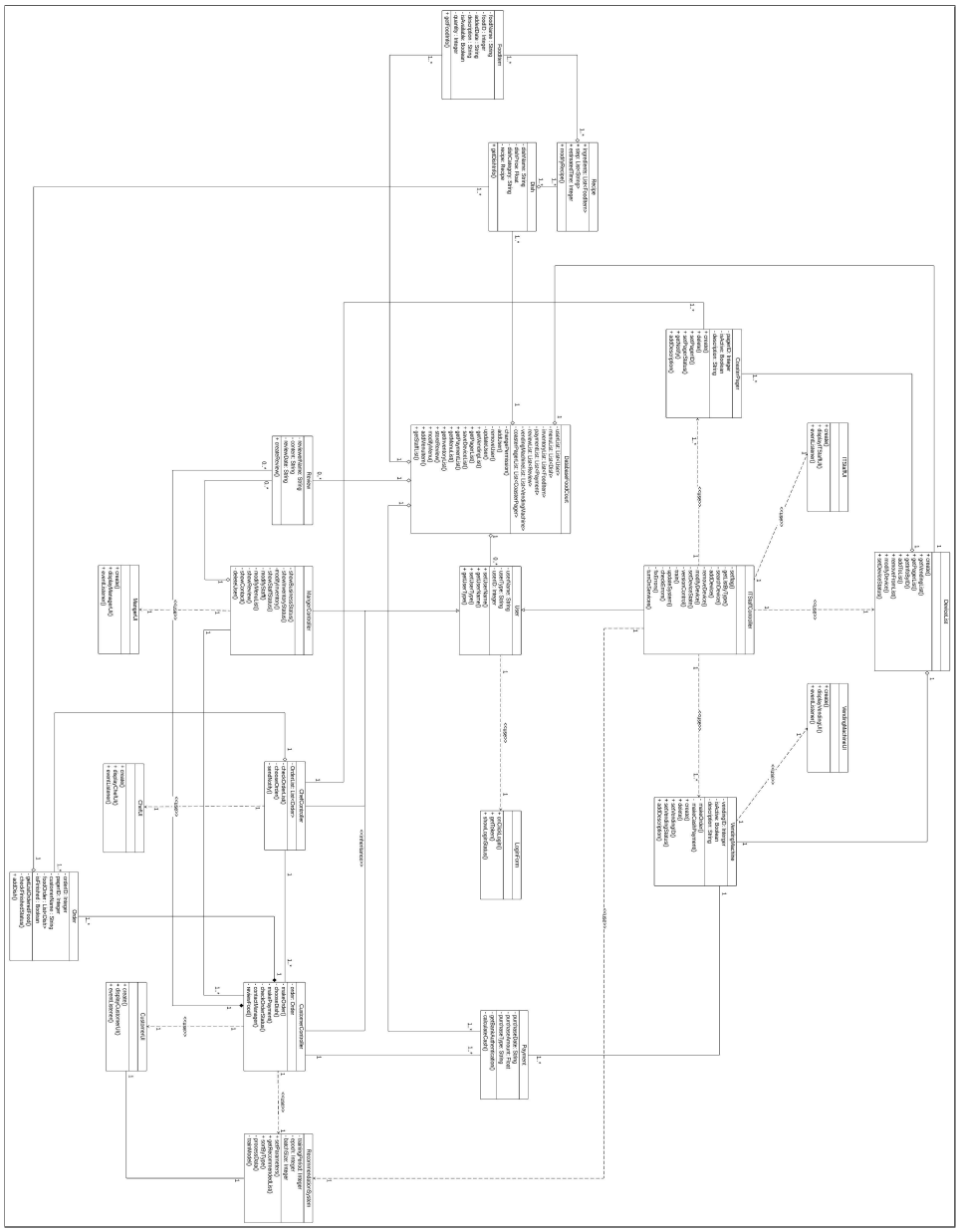

Figure 16: Class diagram of the system

15 Method Description

System UI

Class	Method	Description
ManagerUI	displayManagerUI()	display UI for manager
	eventListener()	take String event as input and interact with Manager Controller
	create()	initialize UI
ChefUI	displayChefUI()	display UI for chef
	eventListener()	take String event as input and interact with Chef Controller
	create()	initialize UI
CustomerUI	displayCustomerUI()	display UI for customer
	eventListener()	take String event as input and interact with Customer Controller
	create()	initialize UI
ITStaffUI	displayITStaffUI()	display UI for ITstaff
	eventListener()	take String event as input and interact with ITStaff Controller
	create()	initialize UI
VendingMachineUI	displayVendingUI()	display UI of vending machine for user
	create()	initialize UI
	eventListener()	take String event as input and interact with Vending Machine

User Service

Class	Method	Description
RecommendationSystem	setParameters()	set parameters for training model
	getRecommendedList()	show recommendation of dishes before customer make payment
	sortByType()	sort the recommended list by the type of dish
	processData()	transfer order of user to data type to process
	trainModel()	obtain data from processing data method and generate the training model according to parameters

Controller

Class	Method	Description
ManagerController	showBusinessStatus()	show current business analytic of food court
	showInventoryStatus()	show current inventory statistic of food court
	showStaffStatus()	show the status of all staff members
	modifyStaff()	modify the staff information specified by ID
	modifyMenuList()	modify the current menu list of food court
	showReview()	show all reviews from customer
	showContact()	show the name and message from customer
	deleteUser()	remove user from the user list
ChefController	checkOrderList()	check ordered list from customer
	chooseOrder()	choose the latest order from customer
	sendNotify()	send notification to customer that the dishes has done
ITStaffController	setflag()	set the condition
	getListByType()	get the devices based on the device type
	searchDevice()	get the device information by type and ID
	addDevice()	add new device to the list device
	removeDevice()	remove device from the list by device type and device ID
	modifyDevice()	change information of device property
	setDeviceState()	change the device operating state
	train()	train the model with data from database and save the recommended list result
	versionControl()	get the system version and latest version
	updateSystem()	upgrade system version to up-to-date
	checkErrors()	scan the system for errors
	fixErrors()	fix all errors in system
	turnOnServices()	turn on services before exit
CustomerController	makeOrder()	make order for customer by mobile application
	chooseDish()	get the dish when users choose dish in the recommended list
	makePayment()	make payment for customer by mobile application
	checkOrderStatus()	show tracking order for customer on mobile application
	contactManager()	show manager information and send message to manager
	reviewFood()	send review to ordered dishes

System Database

Class	Method	Description
DatabaseFoodCourt	changePermission()	update the permission of user account
	addUser()	add new user to user list
	removeUser()	remove user from user list
	updateUser()	update information of user from list
	getVendingList()	return the list of all vending machine
	getPagerList()	return the list of all pager
	saveDeviceList()	save all the changes in the vending and pager list
	getPaymentList()	store history payment from customer and vending machine
	getMenuList()	return all the the dishes in the menu
	getInventorytList()	return list of all food items
	storeReview()	save the customer review
	modifyMenu()	change the information of dish in the menu
	addMenuItem()	add new dish to the menu
	getStaffList()	return the list of all staff members
CoasterPager	create()	initialize an instance if there is a new pager
	delete()	delete the object if the device is not used anymore
	setPagerID()	set coaster pager ID
	setPagerStatus()	set coaster pager status
	getNotify()	vibrate the coaster pager
	addDescription()	add new description to coaster pager
VendingMachine	makeOrder()	return an order list from current customer
	makeCashPayment()	notify customer to put in cash in vending machine
	create()	initialize an instance if there is a new vending
	delete()	delete the object if the device is not used anymore
	setVendingID()	set ID of vending machine
	setVendingStatus()	set vending machine current status
	addDescription()	add new description for vending machine
DeviceList	create()	initialize an instance of the class
	getVendingList()	return a list of vending machines and their active status
	getPagerList()	return a list of coaster pagers and their active status
	getInfoByID()	return the device with ID specified from list
	addToList()	add new vending machine or new coaster pager to list
	removeFromList()	remove a device with ID specified from list
	modifyDevice()	adjust the information of the vending machine or pager in list

Other related methods

Class	Method	Description
Order	getListOrderedFood()	get list order of food from user
	checkFinishedStatus()	check the status of current order finish or not
	addDish()	add the dish chosen by user in the recommended list
FoodItem	getFoodInfo()	return the information of current food
Dish	getDishInfo()	return the information of current dish
Review	createReview()	create a review form with name, content, date
Recipe	modifyRecipe()	modify ingredients, step, estimated time of each dish
User	setUserName()	set name of current user
	getUserName()	get name of current user
	setUserType()	set type of current user (ITstaff, Chef, Customer)
	getUserType()	get type of current user (ITstaff, Chef, Customer)
Payment	getBankAuthentication()	get authentication from bank server
	calculateCash()	calculate the amount of cash user has put in

16 Sequence Diagram At Detail Level

Please zoom in to see more clearly

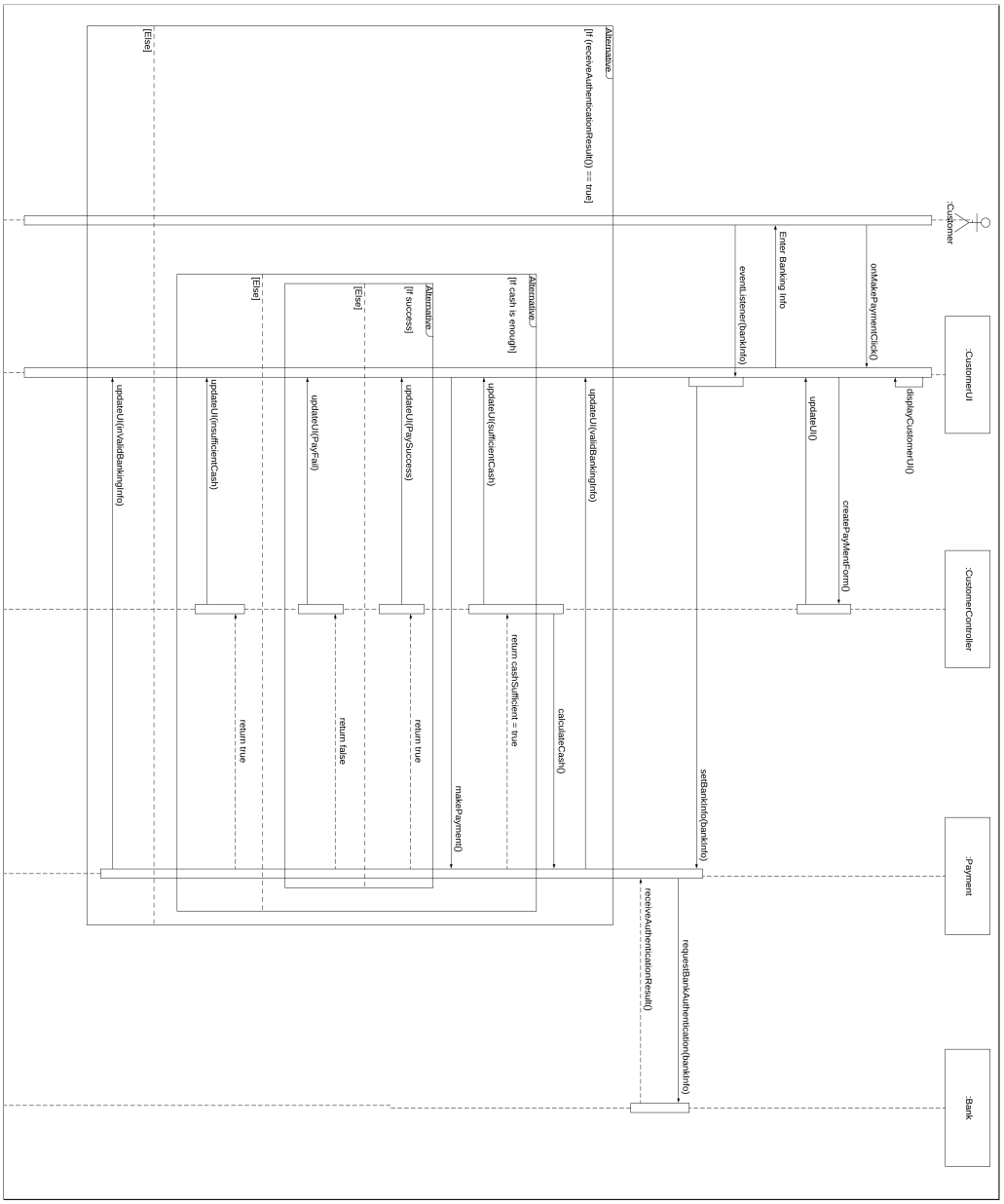

Figure 17: Sequence Diagram of Make Payment feature at detail level

17 Activity Diagram At Detail Level

Please zoom in to see more clearly

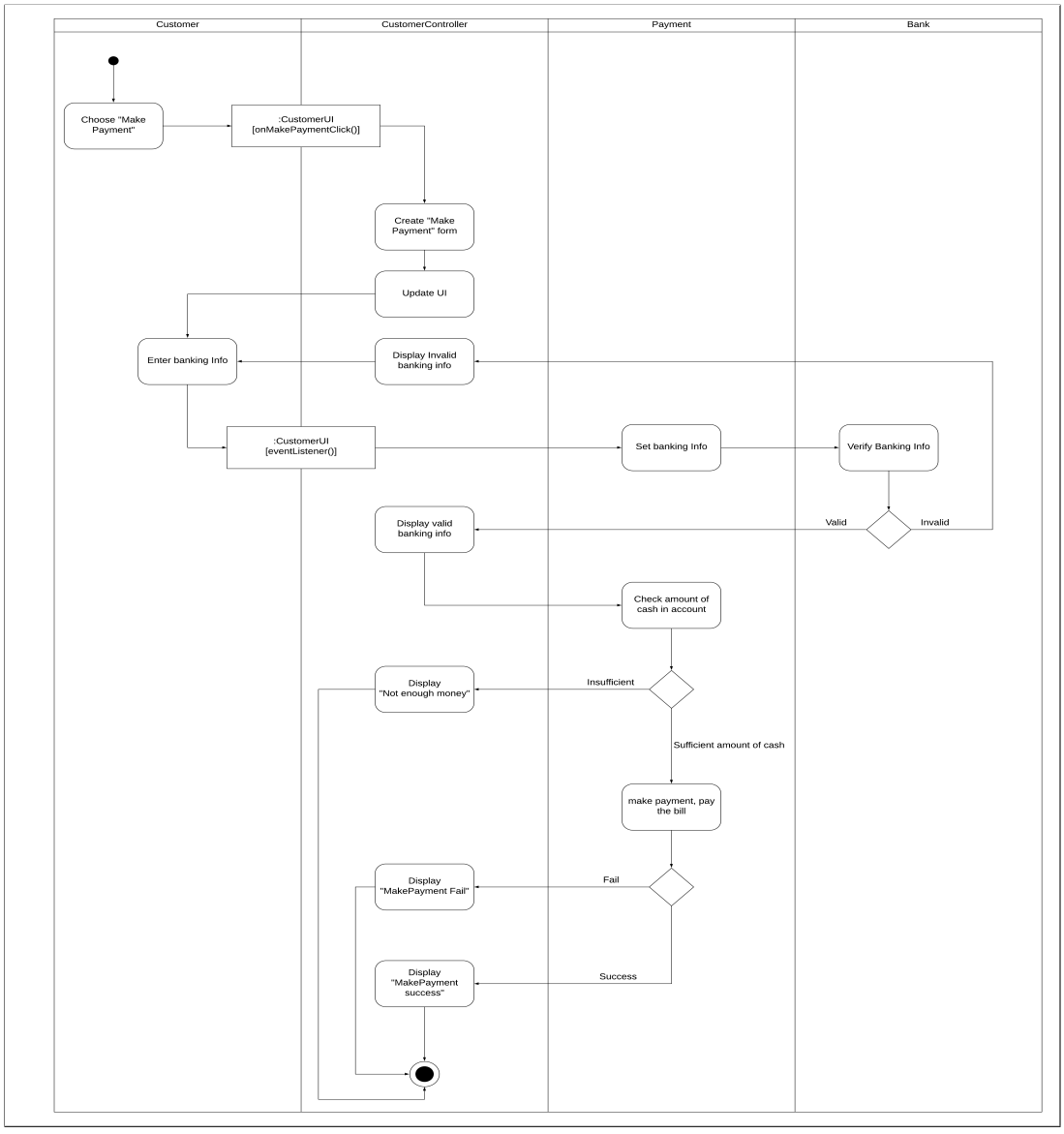

Figure 18: Activity Diagram of Make Payment feature at detail level

18 Design Pattern

In this project we use three common design patterns including **Singleton**, **State** and **Observer**:

-Singleton(Used in classes: DatabaseFoodCourt, DeviceList, RecommendationSystem): Singleton is a basic design pattern. This type of design pattern comes under the creational pattern as this pattern provides one of the best ways to create an object.

This pattern involves a single class which is responsible to create an object while making sure that only a single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class. DatabaseFoodCourt, DeviceList, RecommendationSystem class are initiated one instance only for each class.

-State(Use in classes: VendingMachine and CoasterPager): State pattern is a behavioral design pattern. This pattern is used to encapsulate varying behavior for the same object, based on its internal state.

Because both the VendingMachine and CoasterPager classes have two distinct states representing the working status: active or inactive. And functions can be operated in these two different states.

-Observer(Used in classes: All UI classes): The observer pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

The object which is being watched is called the subject. The objects which are watching the state changes are called observers or listeners. All the UI classes are the subject and the controller classes, DatabaseFoodCourt and other classes were changed by UI action are listener

19 A working demonstration

Please zoom in to see more clearly

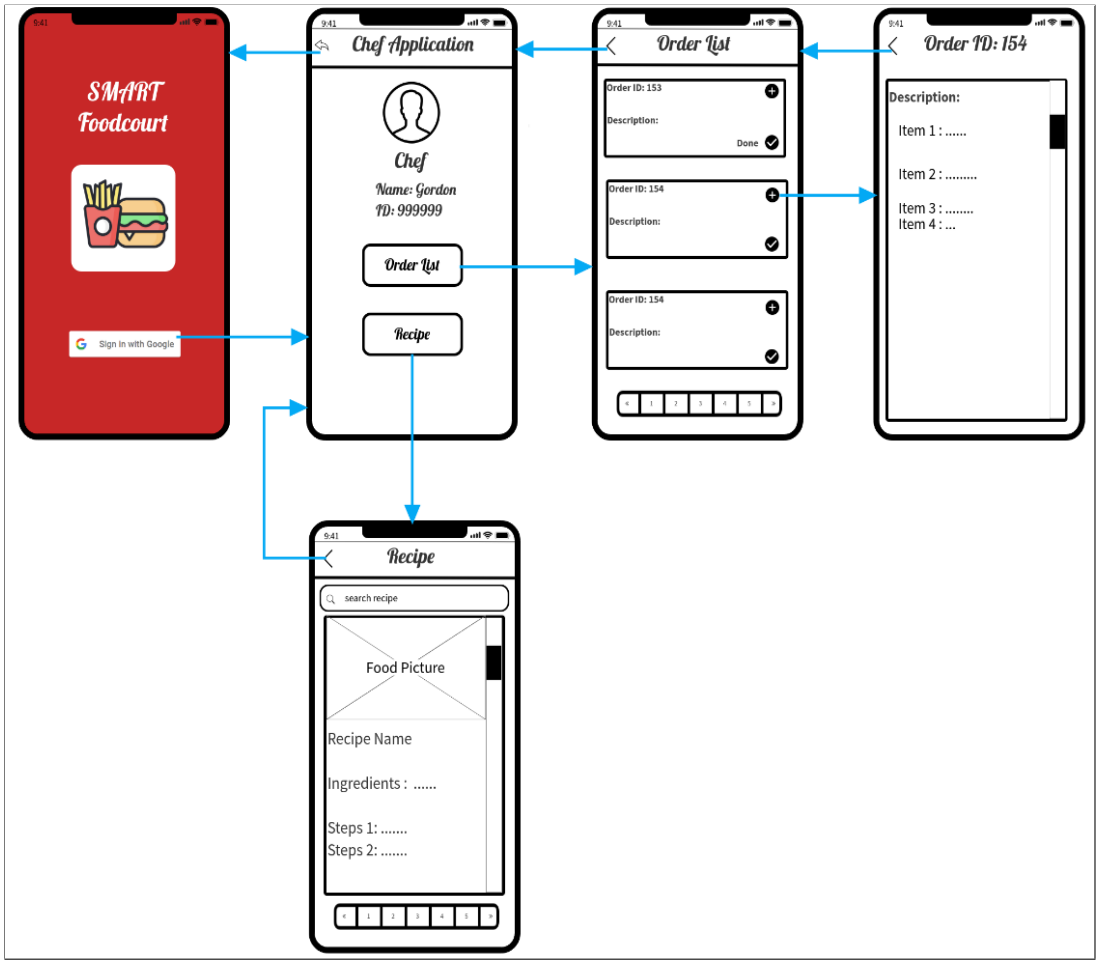

Figure 19: Screens of the application for Chef