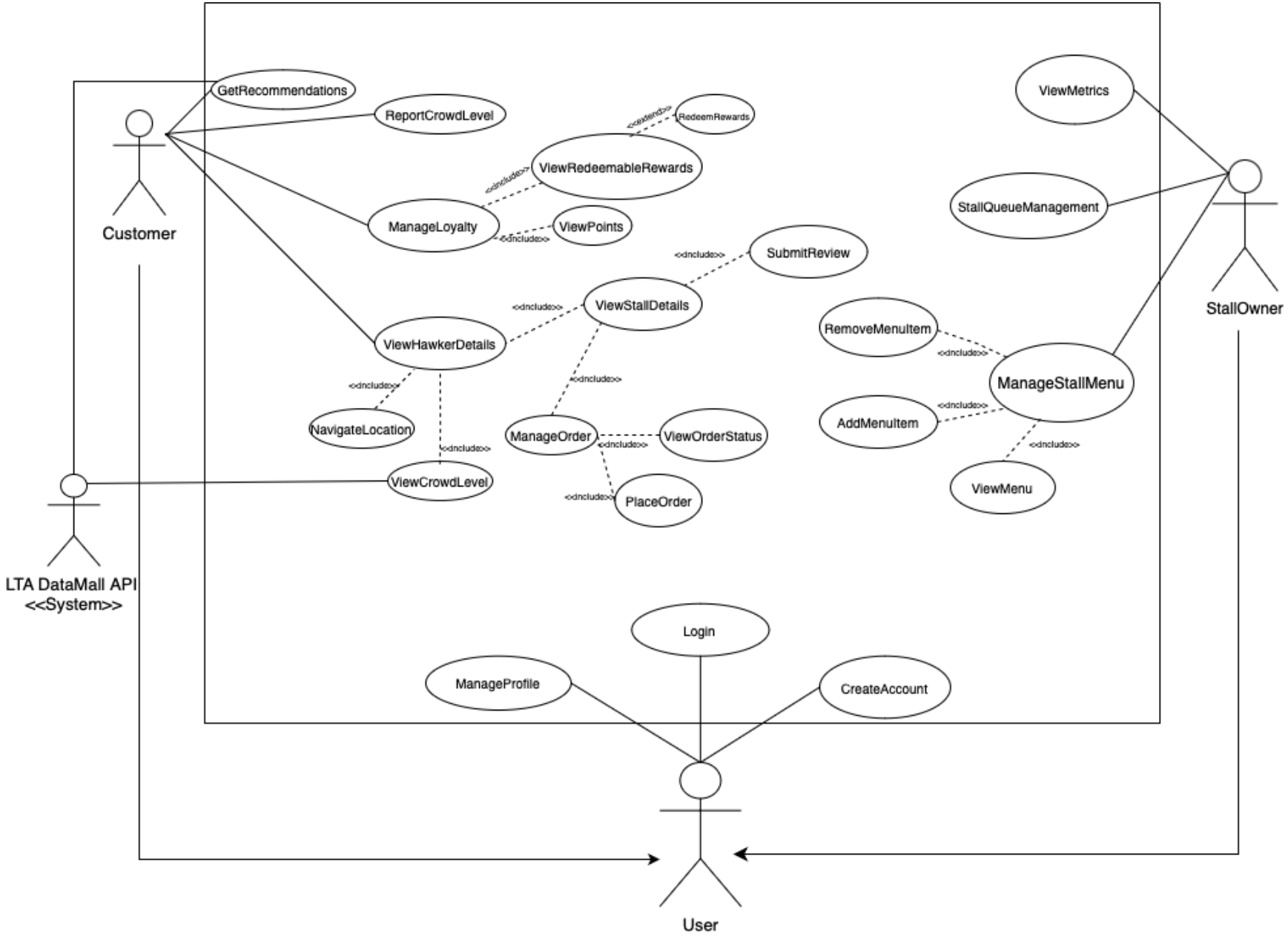
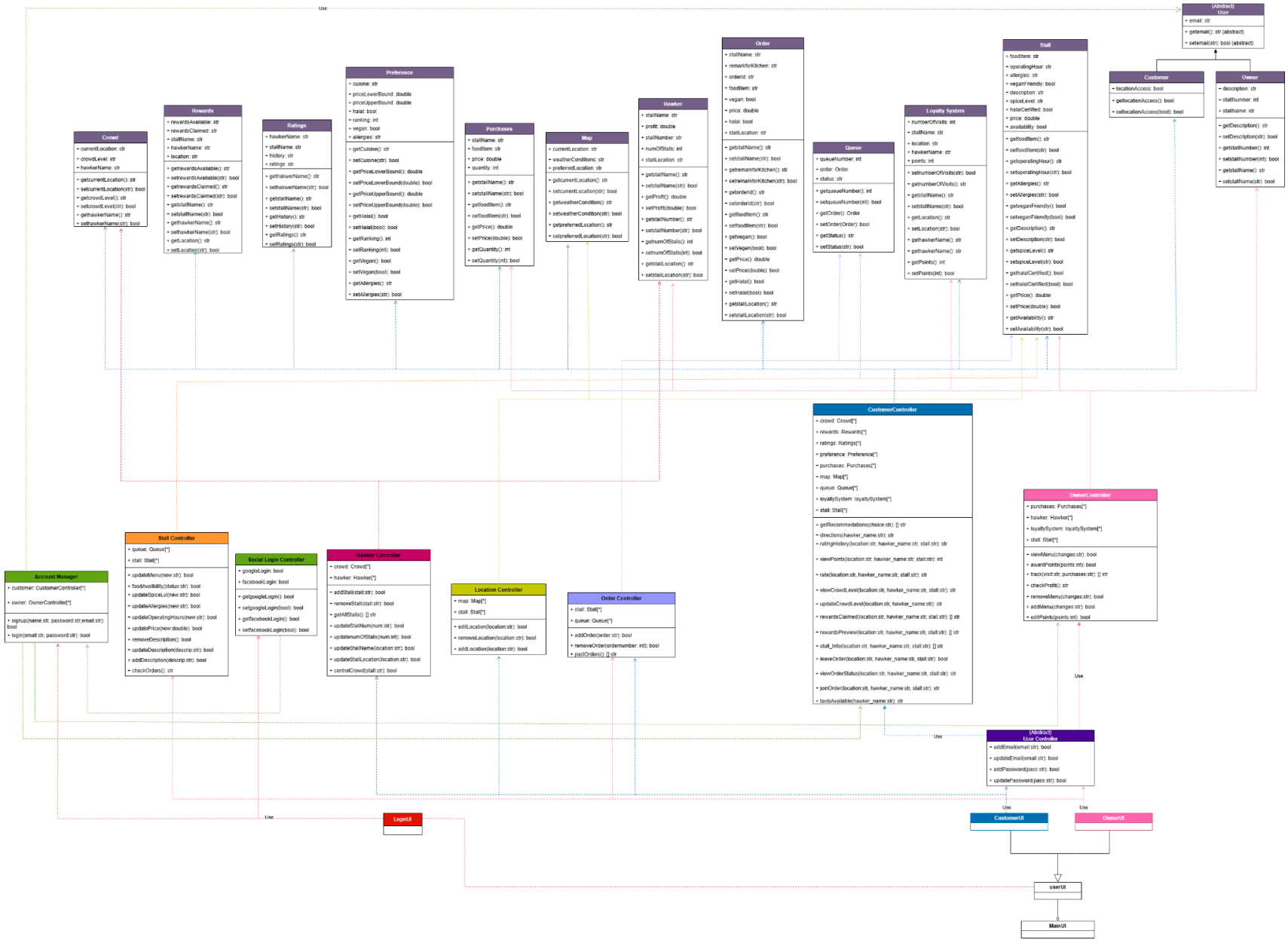
SC2006 - Software Engineering Lab 3 Deliverables

Lab Group	**SCEB**
Team	**Team 3 (HawkerGo)**
Members	**Dosi Veer (U2323005K)**
	Razan Siraj (U2321259G)
	Chin Jiaqi (U2421026J)
	Jain Chetan Ishita (U2321668B)
	Le Nguyen Bao Huy (U2322337G)
	Liong Xun Qi (U2322609H)

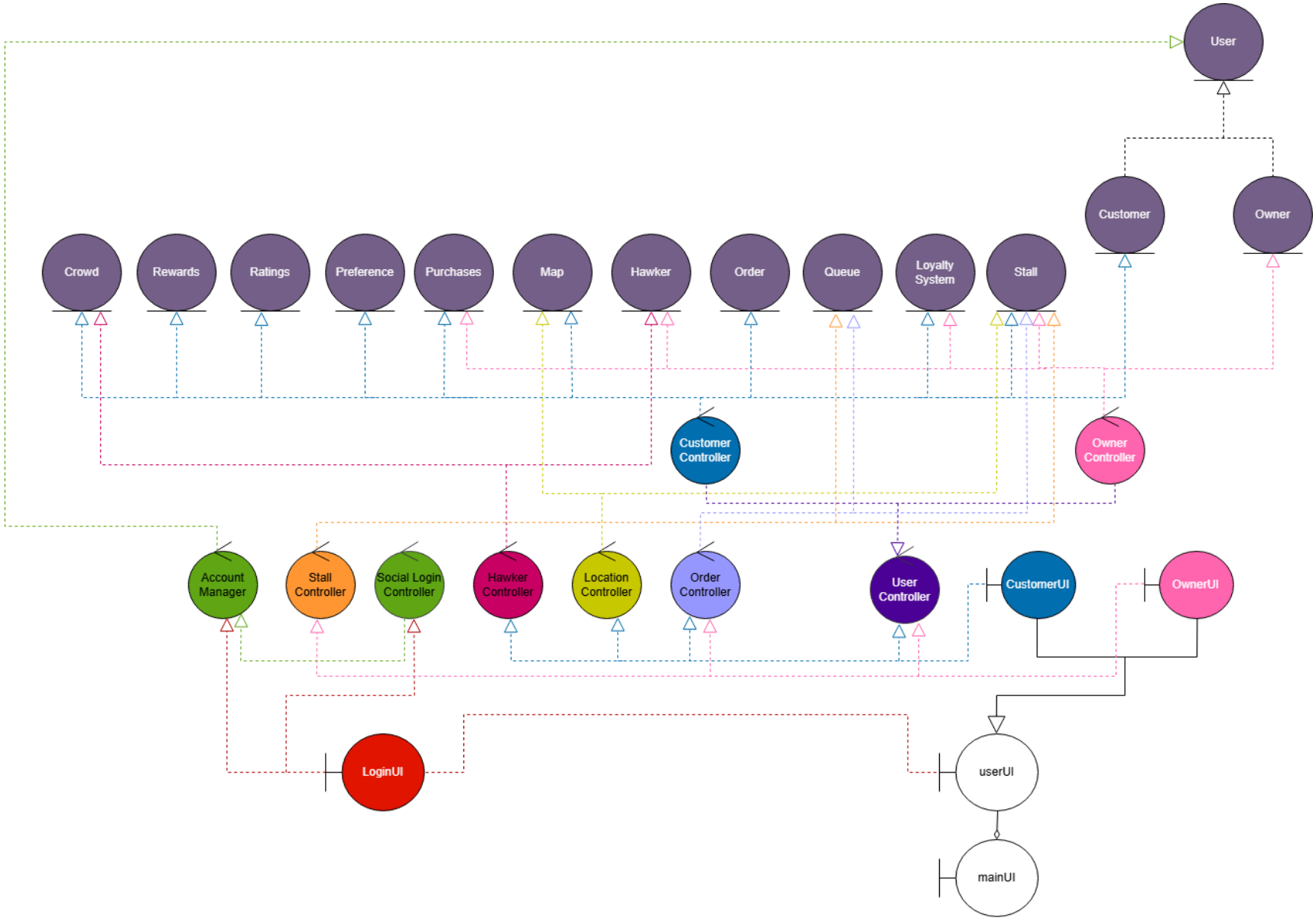
1. Use case diagram



2. Key Classes Diagram



3. Entity, Boundary, Control Diagram



4. Design Model

We will be using MVC architecture based on the Models, UIs and Controllers defined in the Entity, Boundary and Control Diagram. To enhance extensibility and maintainability, we will be adhering to good software design principles, such as Loose Coupling, Single Responsibility, Dependency Inversion and Interface Segregation Principles. Some Design Patterns we will be implementing include:

Factory Pattern: This means the exact class of the object created is only known during runtime. It encapsulates the instantiation logic and allows for more flexibility and decoupling of code. It is implemented in the creation of either Customer or StallOwner instances upon Login during run time. It is an implementation of the Liskov Substitution Principle.

Observer: Observer Pattern defines one to many dependencies between objects. When the object changes state, all its dependencies will automatically be notified and updated.

This will be implemented in the ManageQueue, ManageOrder, or ViewOrderStatus classes to ensure customers receive live updates when order status is changed or when queue status updates.

Loose coupling is especially beneficial for the observer pattern. It reduces dependencies between subjects and observers, whilst it enhances maintainability, flexibility and scalability.

Singleton: We will use this design pattern to ensure that only one instance of a database connection is instantiated per User. This will prevent API Overload and improve performance.

Facade: The Facade Pattern is applied when the controllers serve as interfaces for interacting with the application's business logic, acting as entry points to the application. It is used for all the Controller Classes. This helps enforce Single Responsibility and Interface Segregation Principles.

User Interfaces (UI)

User Interfaces are what the users see and interact with on the application.

- **MainUI:** This is a high-level interface that other UIs are derived from.
- **UserUI**: This is the abstract interface from which LoginUI, CustomerUI and StallOwnerUI are derived from. It is the UI for users who are not yet logged in.. Users can perform tasks which can be classified under authentication. For example: Users can login through the LoginUI.
- **CustomerUI:** Screens specific only to Customers. For example: Screens to report crowd levels, get recommendations, manage loyalty, and view hawker details.
- **LoginUI:** It comes under UserUI and allows users to login.

Controllers (Facade Pattern)

There are multiple controllers, depending on the specific use case and role of the user.

Account Manager

- Manages user authentication and profile settings.
- Handles password changes, profile updates, and account deletion.
- Works closely with AuthController and SocialLoginController.
- Has access to User authentication data.

Stall Controller

- Manages hawker stalls, allowing stall owners to update their menu, view orders, and manage queues.
- Has access to Stalls, Orders, and Queue Management entities.

Social Login Controller

- Handles third-party authentication using Google, Facebook, etc.
- Works with AuthController to retrieve user profile details.
- Ensures secure authentication workflows.

Hawker Controller

- Manages the hawker-related data, including stall details, menu items, and reviews.
- Supports actions like ViewHawkerDetails, SubmitReview, and ManageOrder.
- Has access to Stalls, Reviews, and Orders entities.

Location Controller

- Manages navigation and location-based services.
- Works with LTA DataMall API to get real-time location updates.
- Supports actions like NavigateLocation and ViewCrowdLevel.

Order Controller

- Handles order processing, including placing orders, updating order status, and viewing orders.
- Works with Queue Management System to ensure efficient order tracking.
- Has access to Orders and Queue entities.

User Controller (High-Level Interface)

- A parent interface from which CustomerController and OwnerController are derived.
- Provides general user functionalities, such as profile management, order tracking, and reporting.

Customer Controller

- Handles customer-specific interactions, including:
    - Reporting crowd levels.
    - Viewing loyalty points and redeeming rewards.
    - Managing food recommendations.
    - Viewing and placing orders.
- Has access to Orders, Loyalty System, and Reviews.

Owner Controller

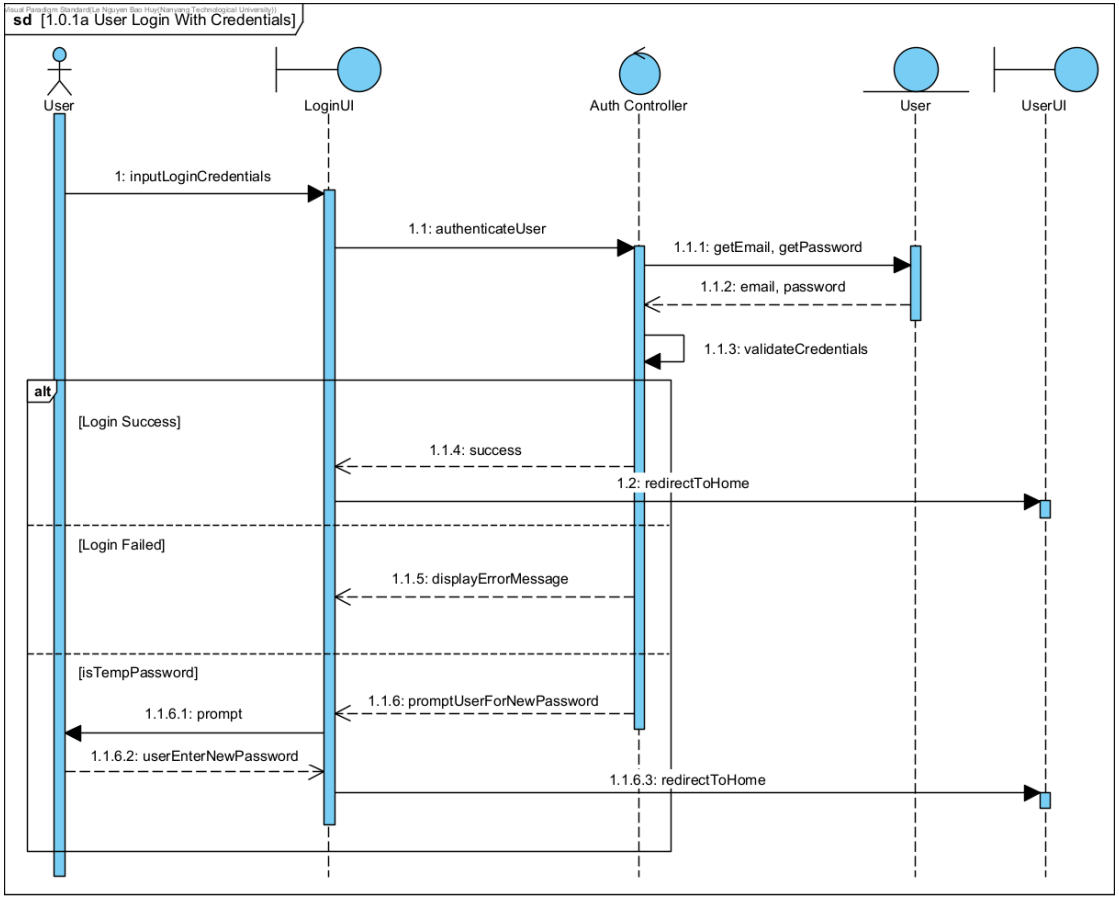
- Manages stall operations, including:
    - Menu management (Add/Remove Items).
    - Queue management (Updating order statuses).
    - Viewing sales metrics.
- Has access to Stalls, Orders, and Analytics.

5. Sequence Diagrams

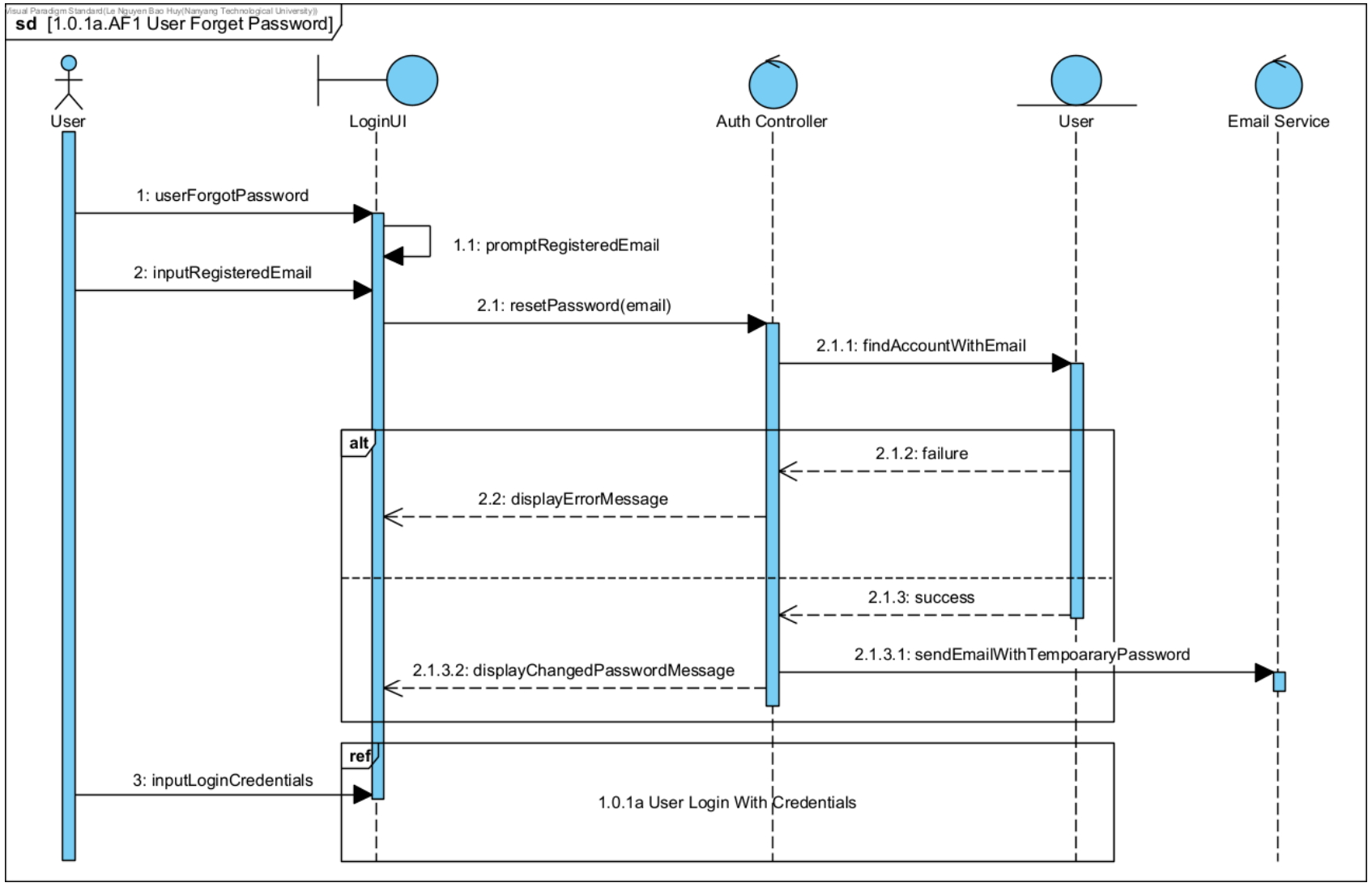
1.0 User Authentication:

1.0.1 User Login:

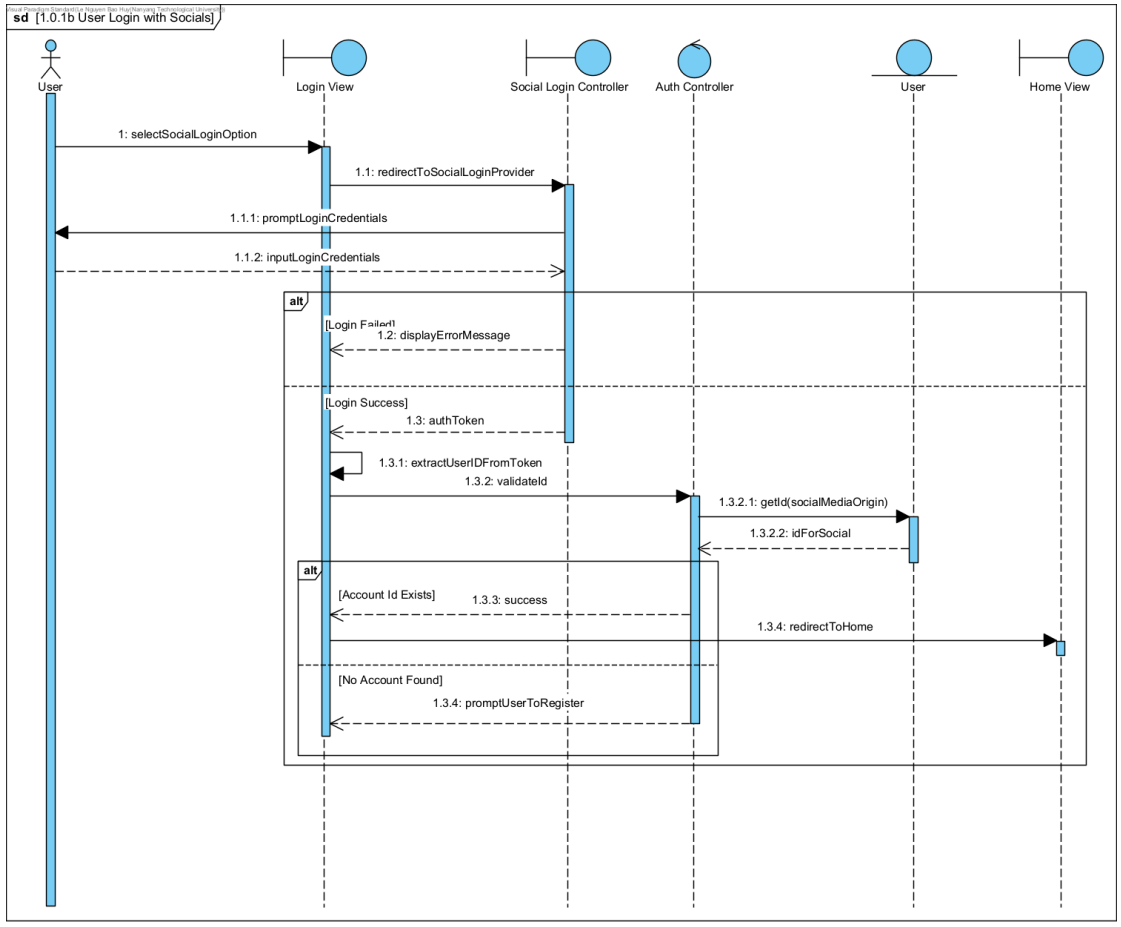
1.0.1a User Login with Credentials



Alternate Flow: User Forgot Password

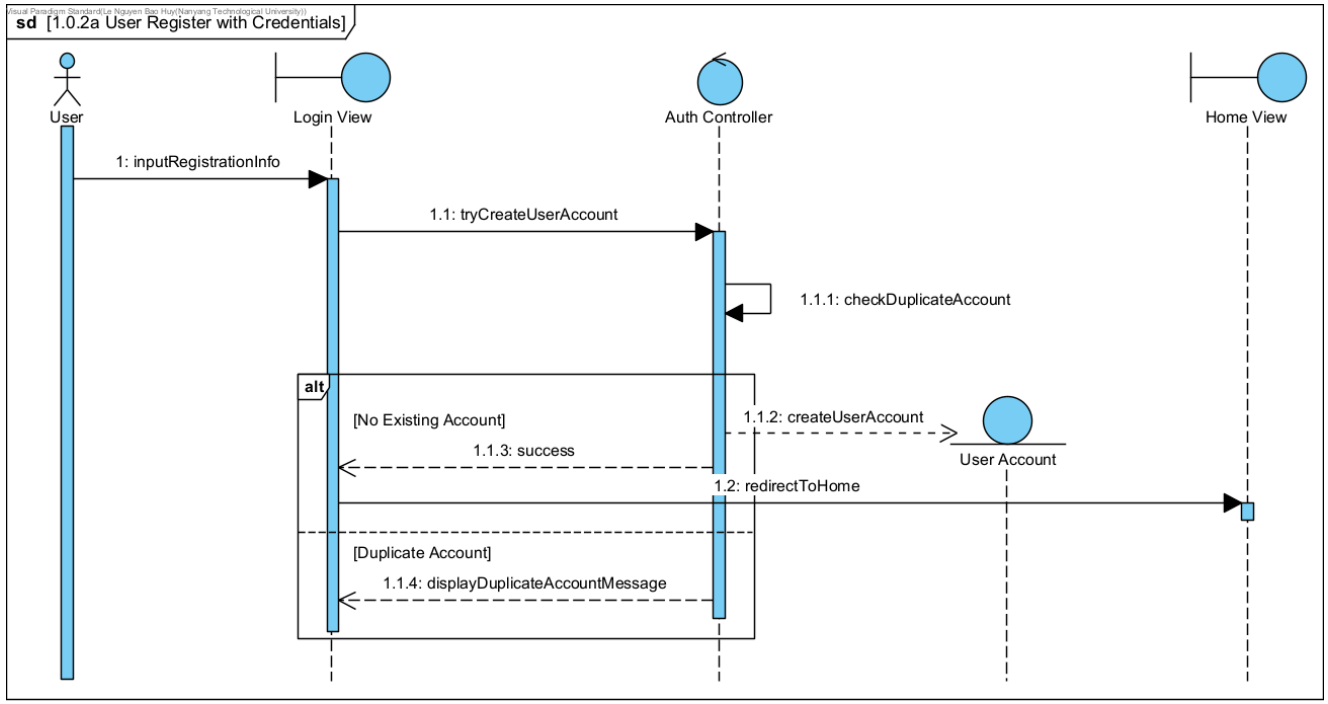


1.0.1b User Login with Socials

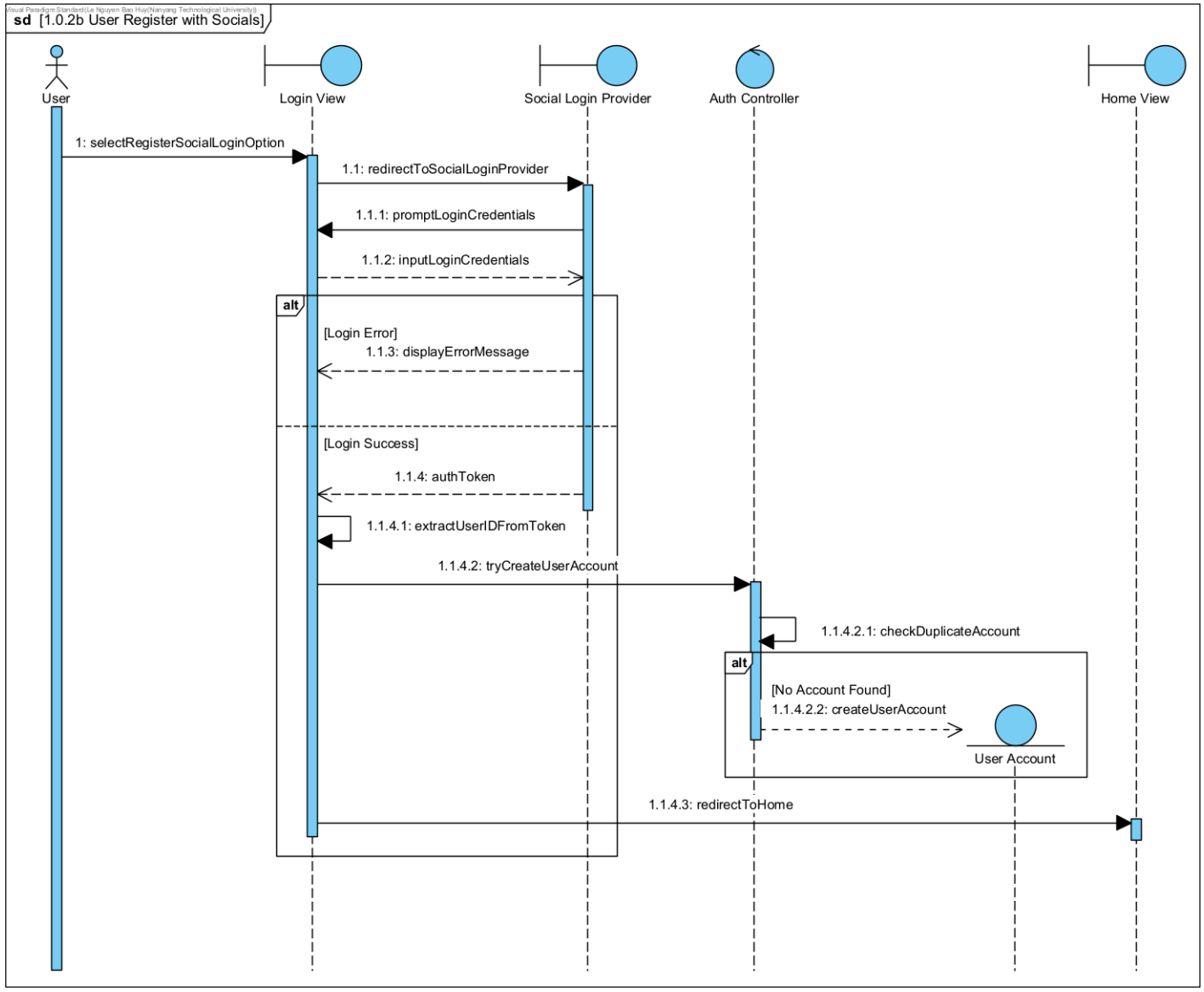


1.0.2 User Registration

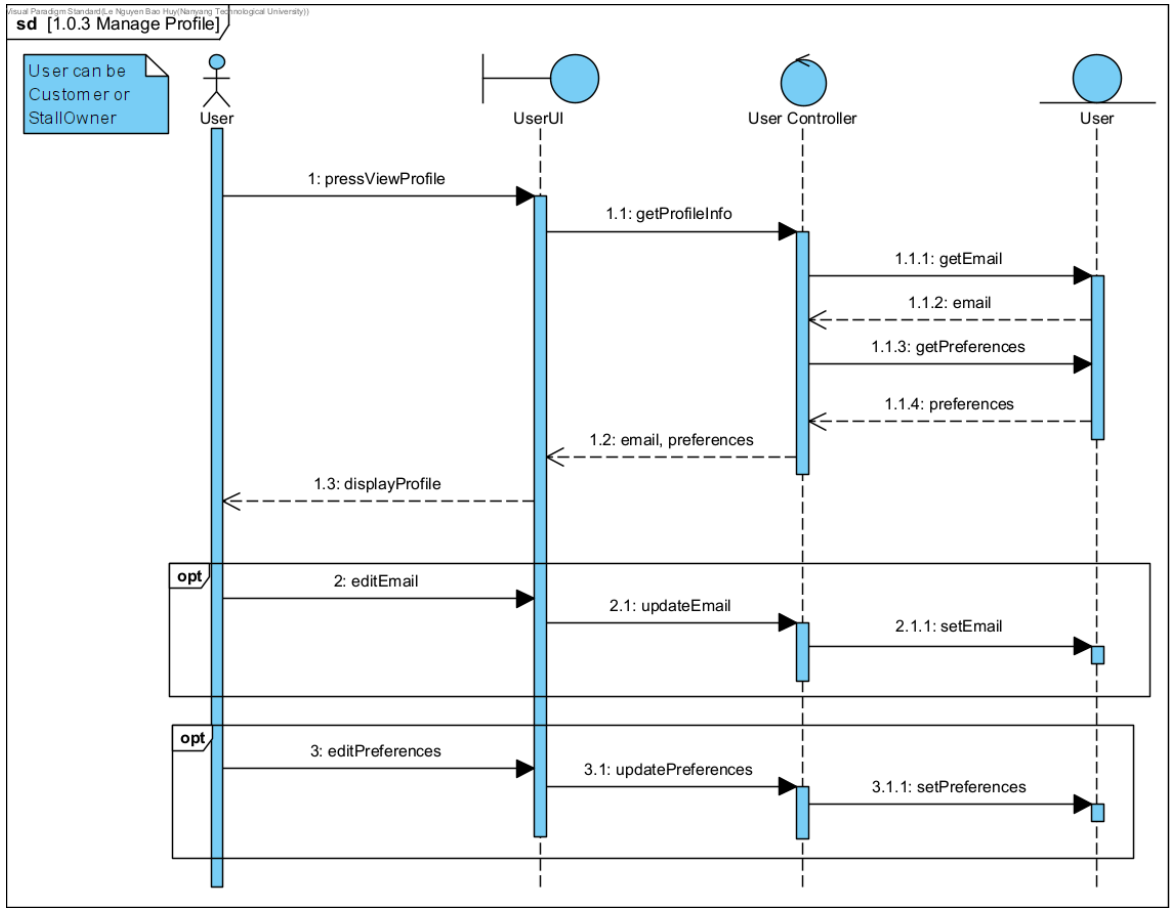
1.0.2a User Register with Credentials



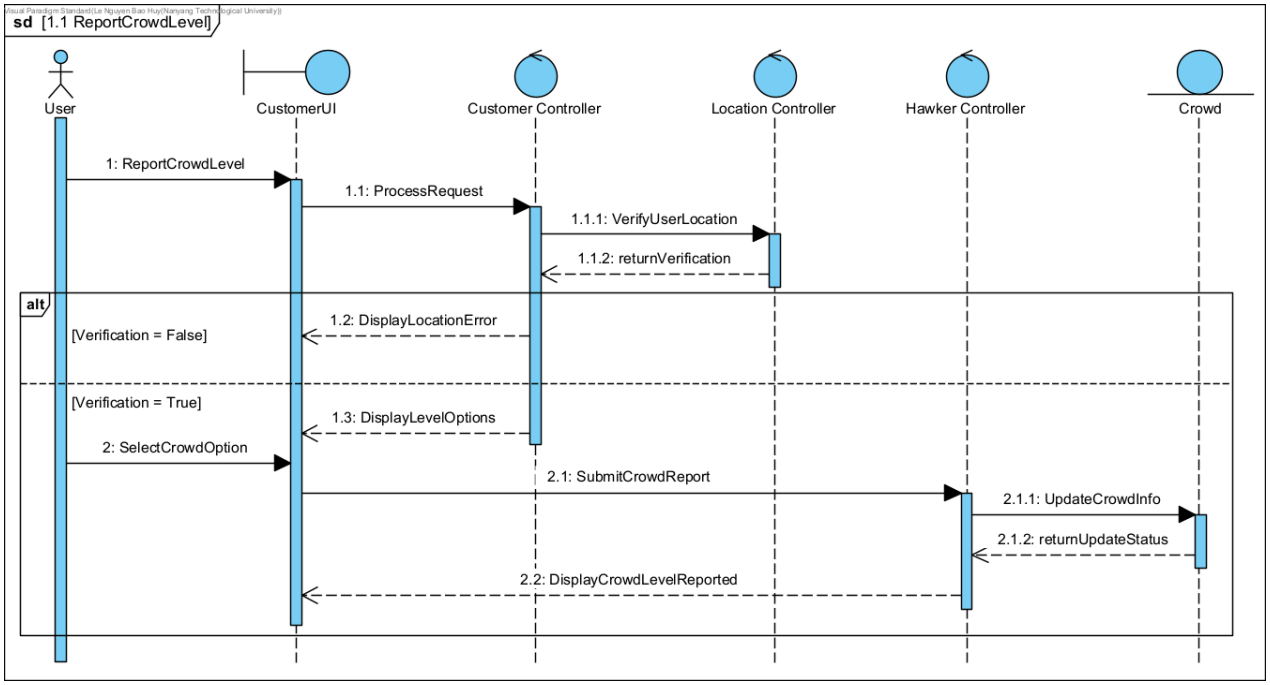
1.0.2b User Register with Socials



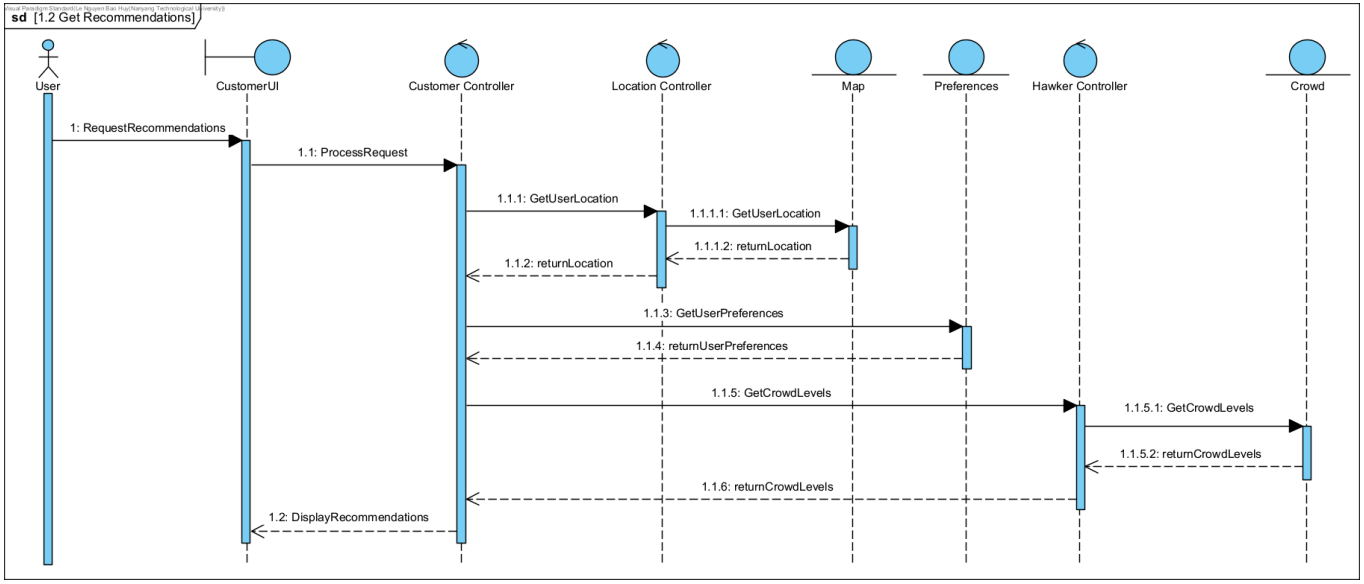
1.0.3 User Profile Management



1.1 ReportCrowdLevel

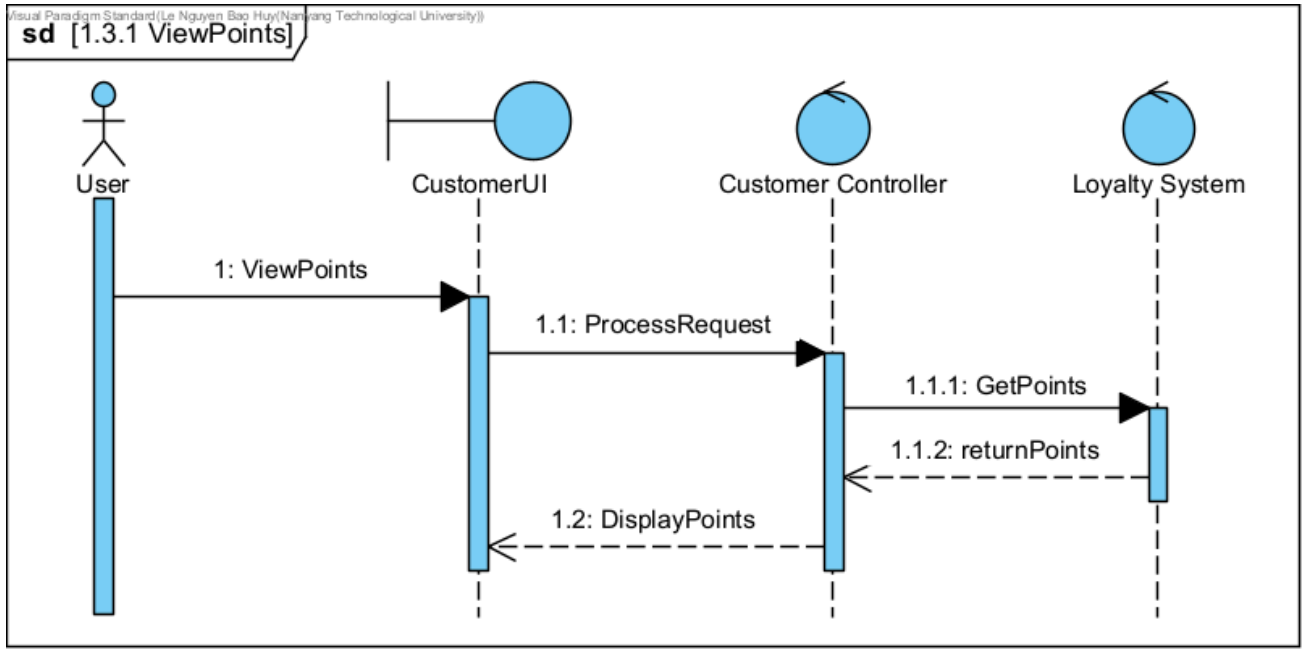


1.2 Get Recommendations

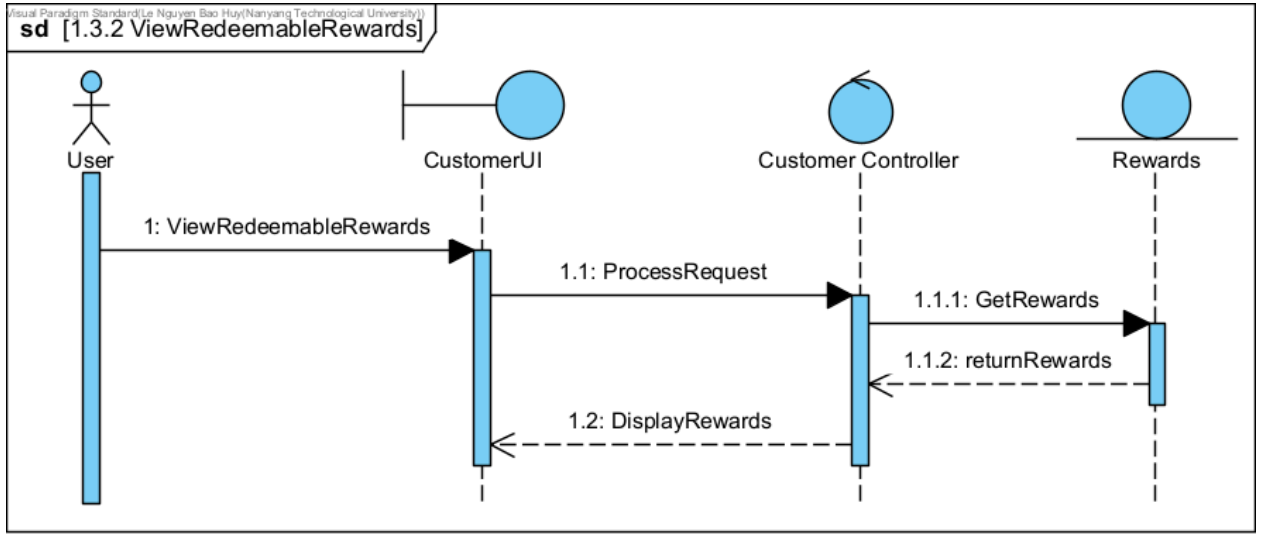


1.3 ManageLoyalty

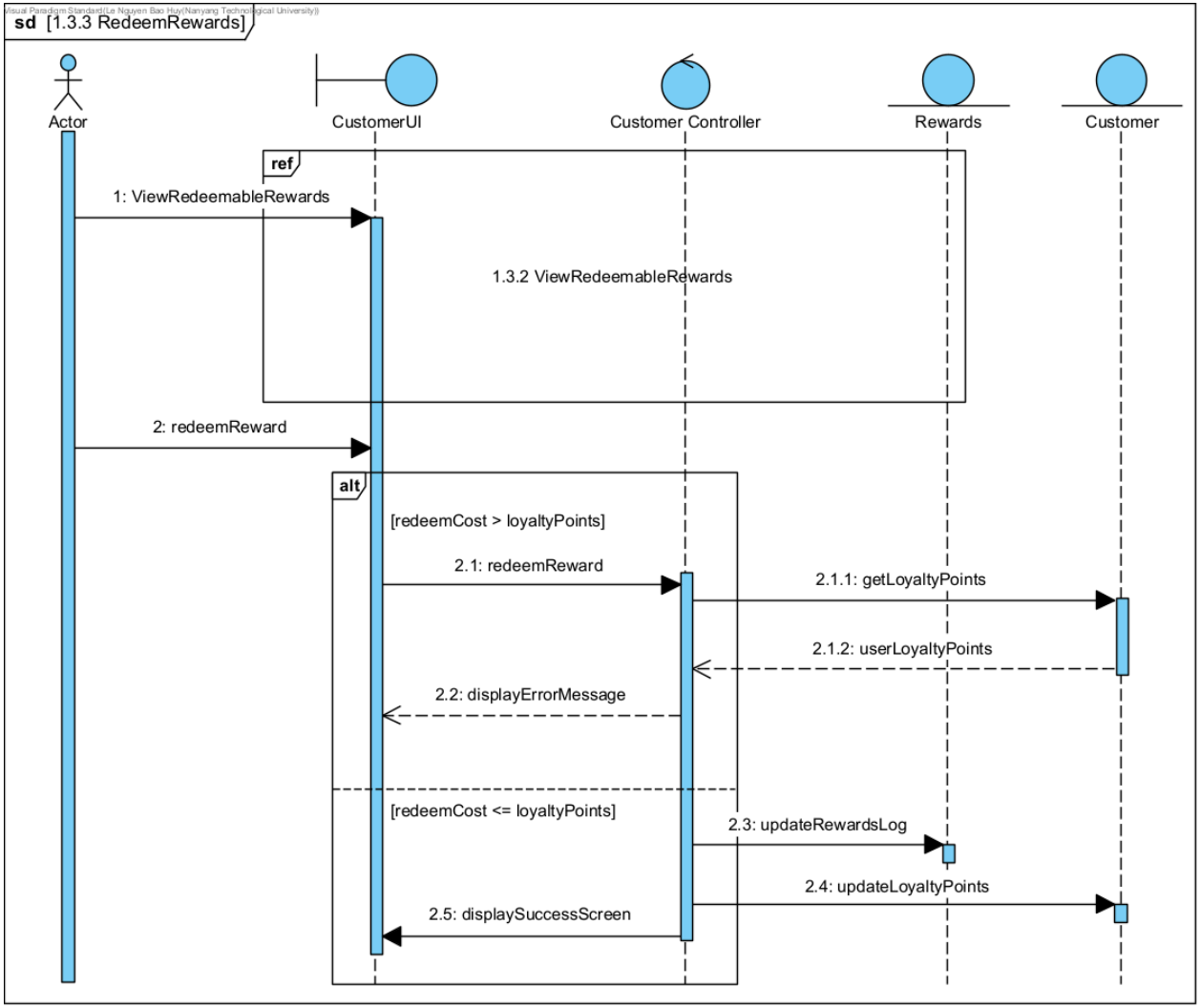
1.3.1 ViewPoints



1.3.2 ViewRedeemableRewards

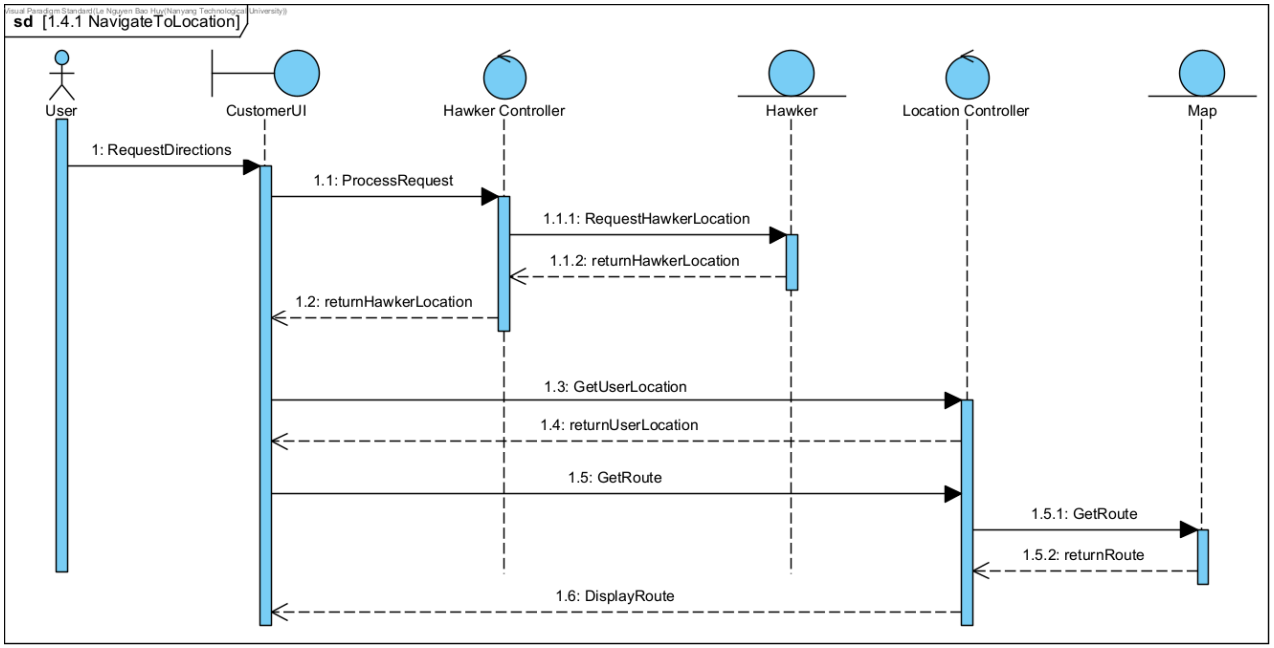


1.3.3 RedeemRewards

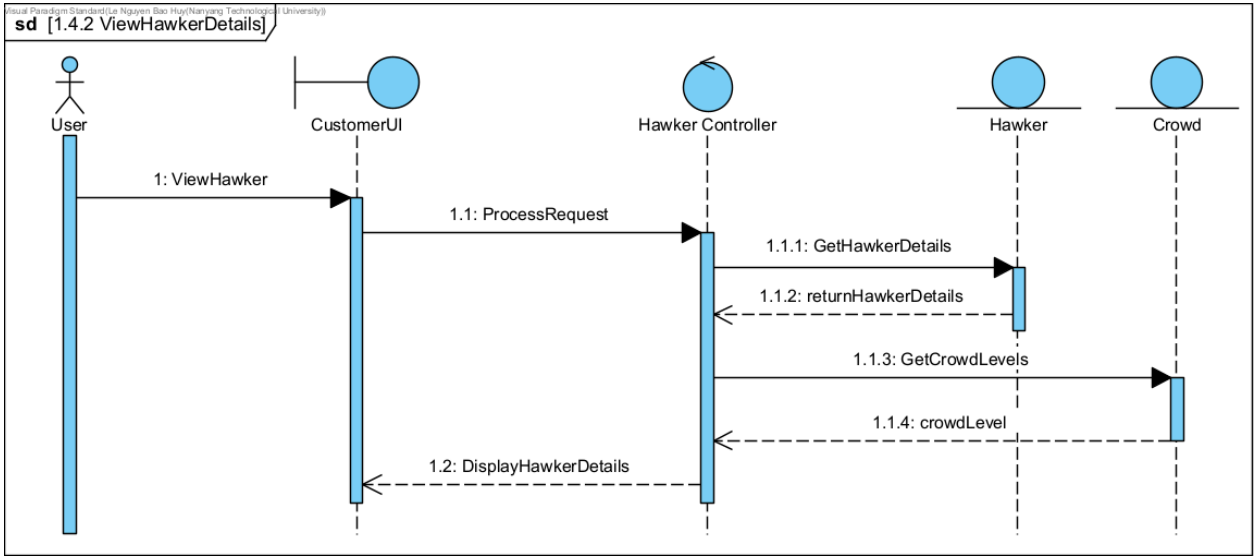


1.4 ViewHawker

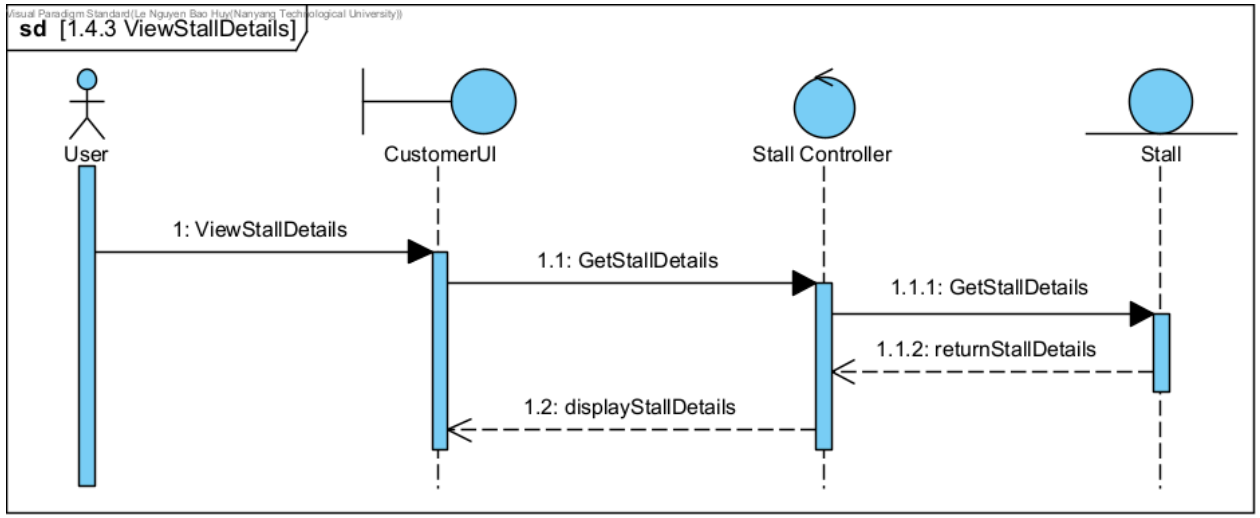
1.4.1 NavigateToLocation



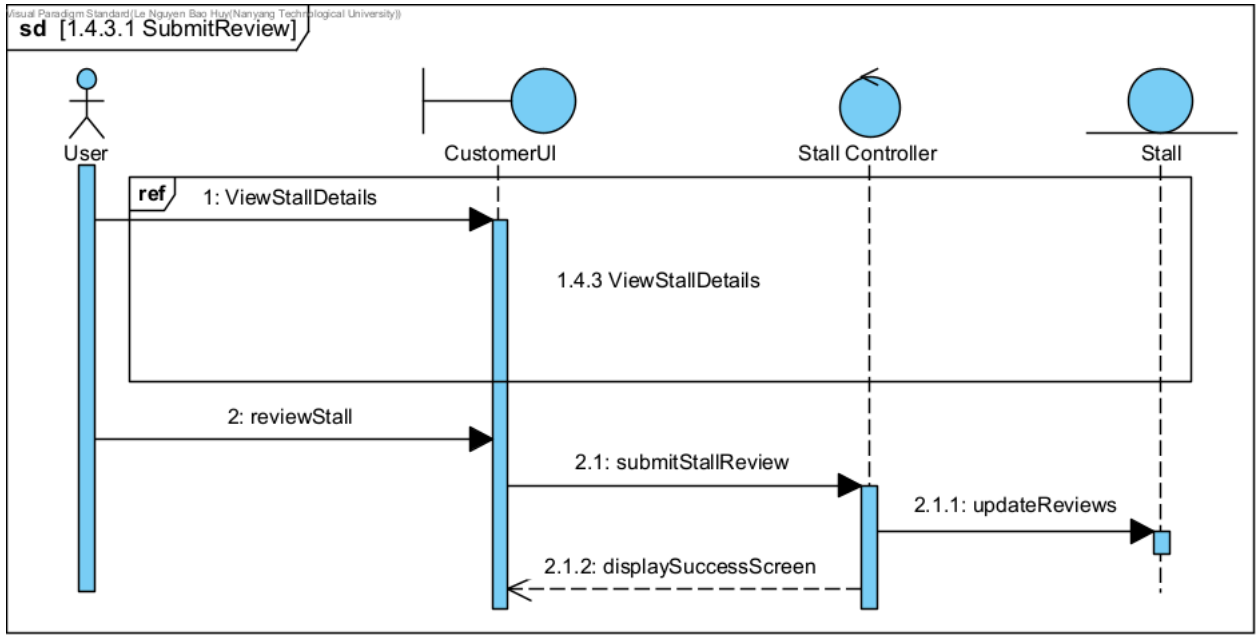
1.4.2 ViewHawkerDetails



1.4.3 ViewStallDetails

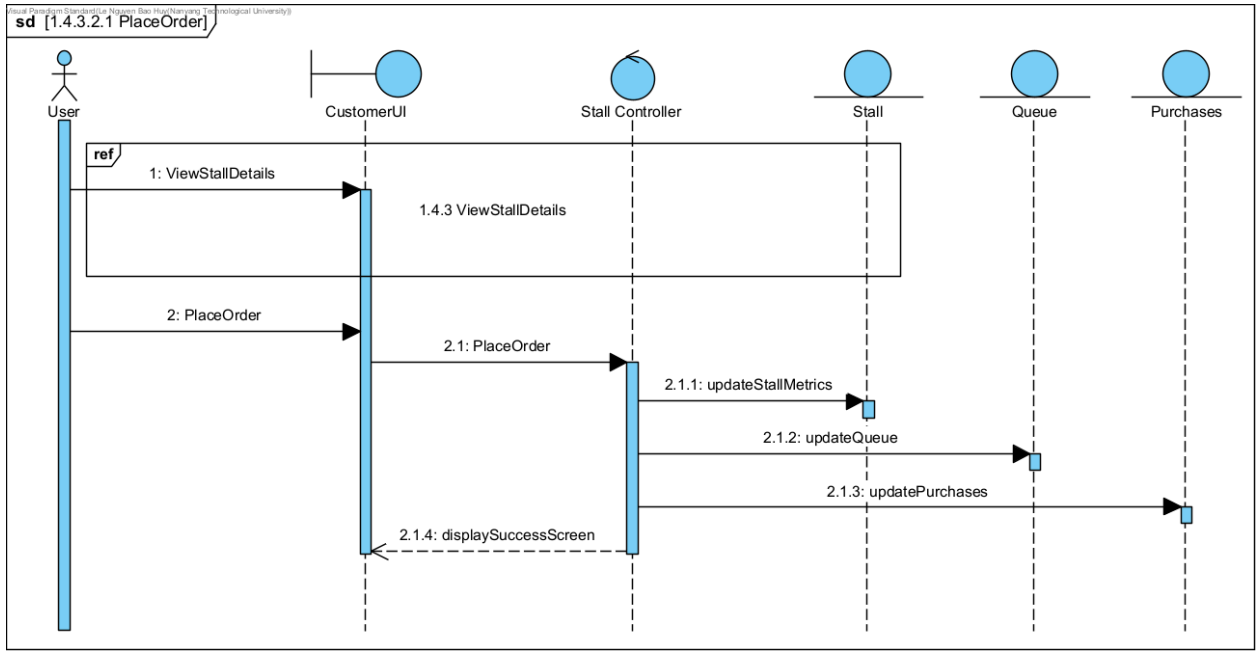

Visual Paradigm Standard(Le Nguyen Bao Huy(Nanyang Technological University))
sd [1.4.3 ViewStallDetails]
User
CustomerUI
Stall Controller
Stall
1: ViewStallDetails
1.1: GetStallDetails
1.1.1: GetStallDetails
1.1.2: returnStallDetails
1.2: displayStallDetails

1.4.3.1 SubmitReview

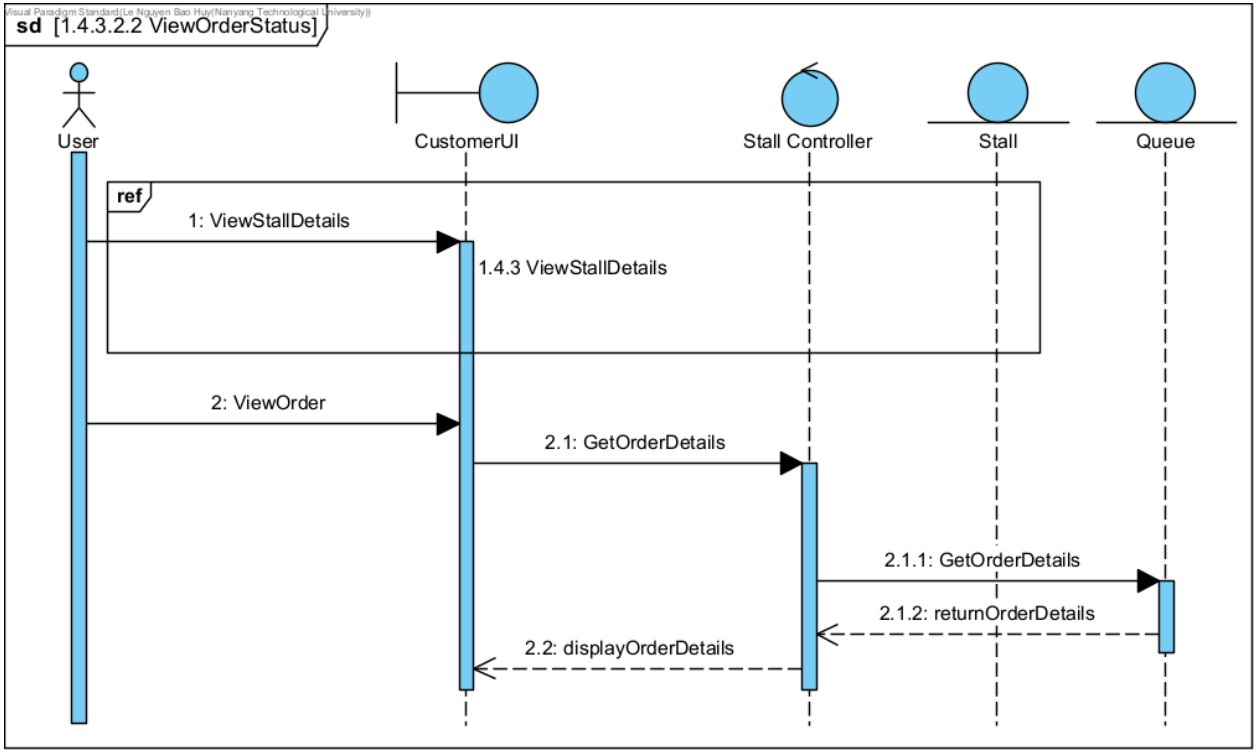


1.4.3.2 ManageOrder

1.4.3.2.1 PlaceOrder

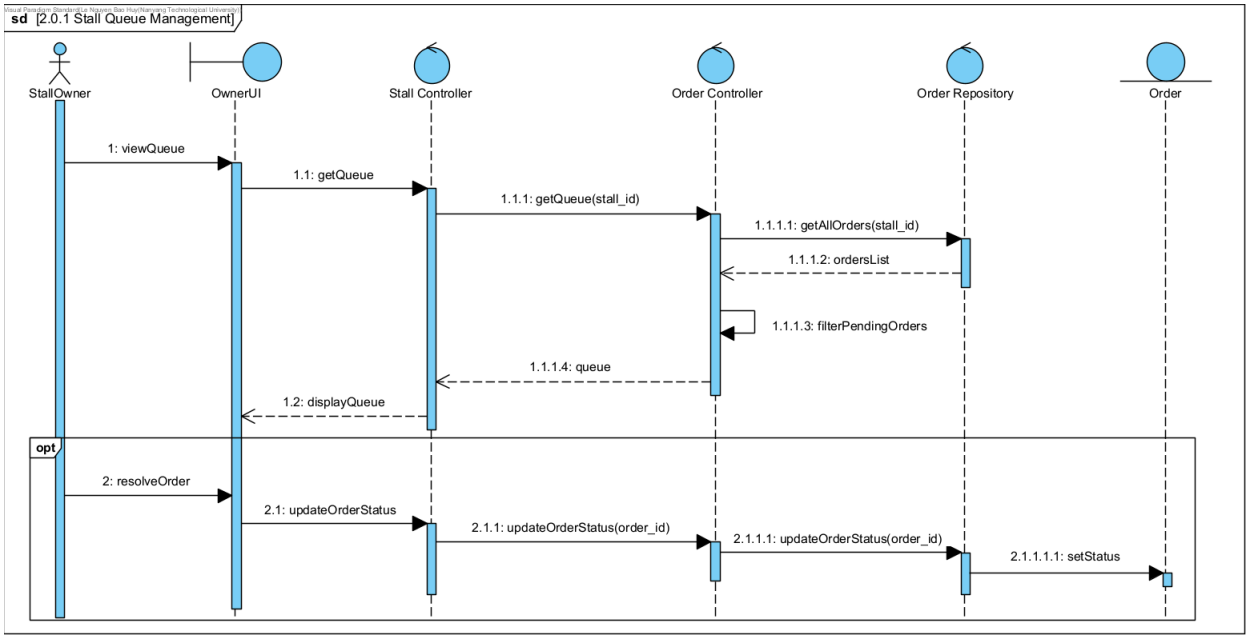

Visual Paradigm Standard(Le Nguyen Bao Huy/Nanyang Technological University))
sd [1.4.3.2.1 PlaceOrder]
User
CustomerUI
Stall Controller
Stall
Queue
Purchases
ref
1: ViewStallDetails
1.4.3 ViewStallDetails
2: PlaceOrder
2.1: PlaceOrder
2.1.1: updateStallMetrics
2.1.2: updateQueue
2.1.3: updatePurchases
2.1.4: displaySuccessScreen

1.4.3.2.2 ViewOrderStatus

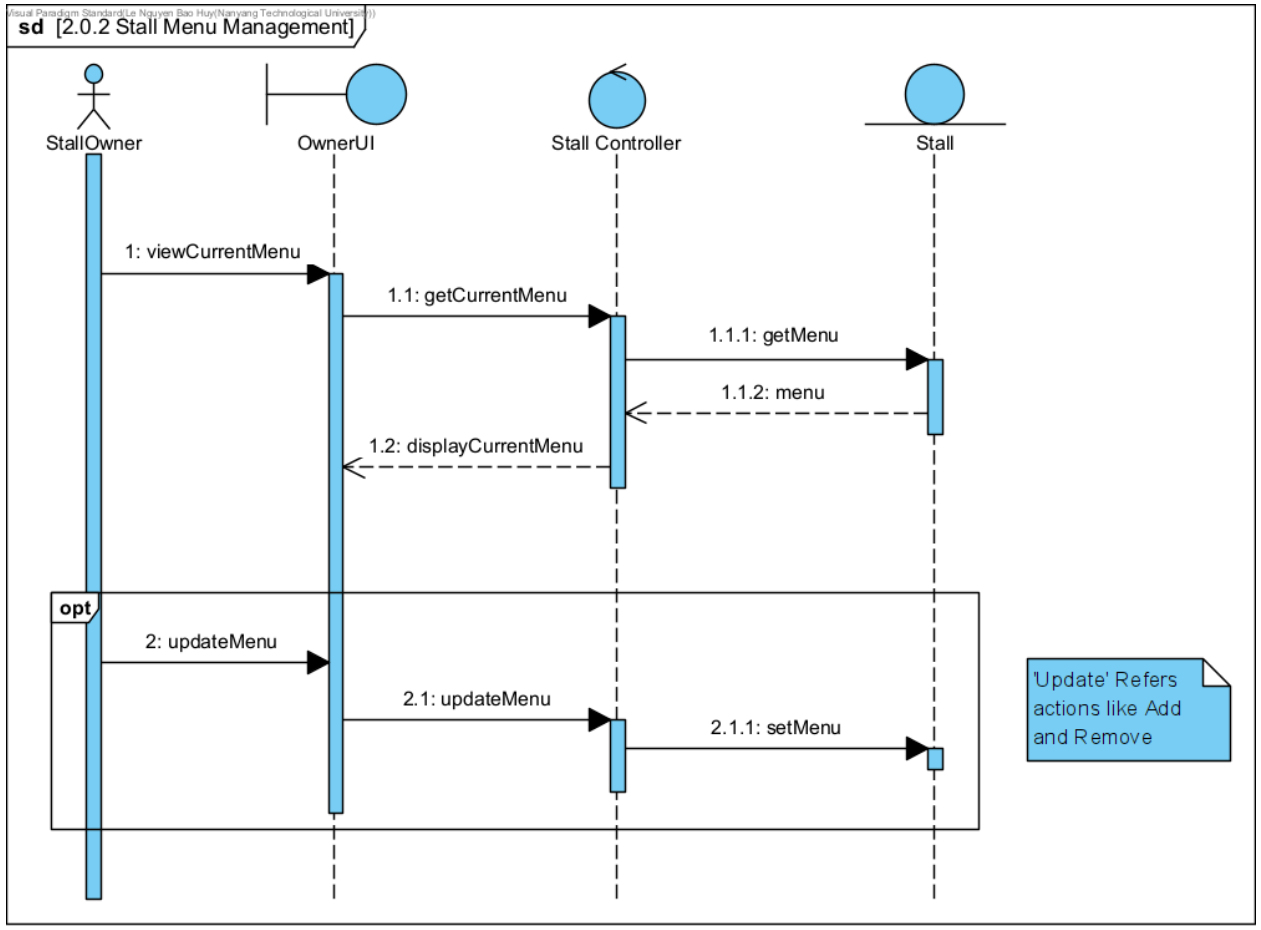


2.0 Stall Operation

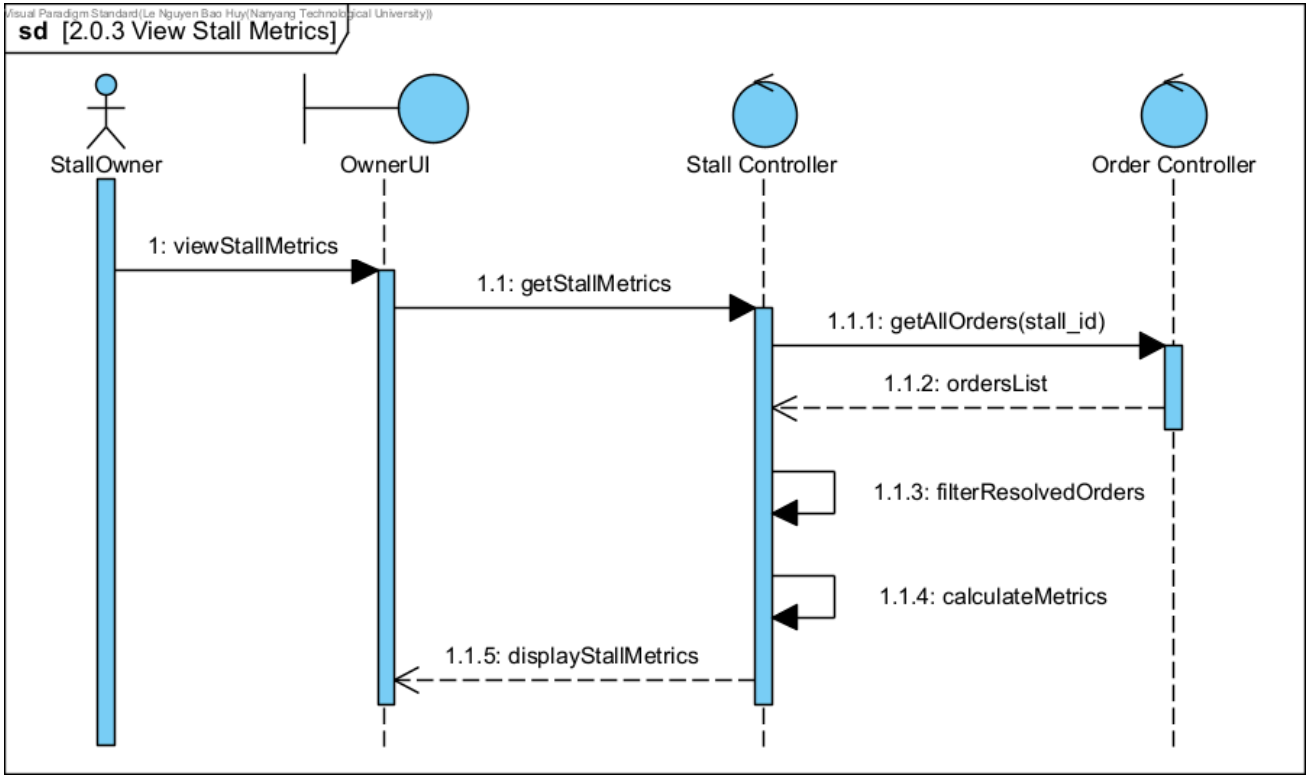
2.0.1 Stall Queue Management



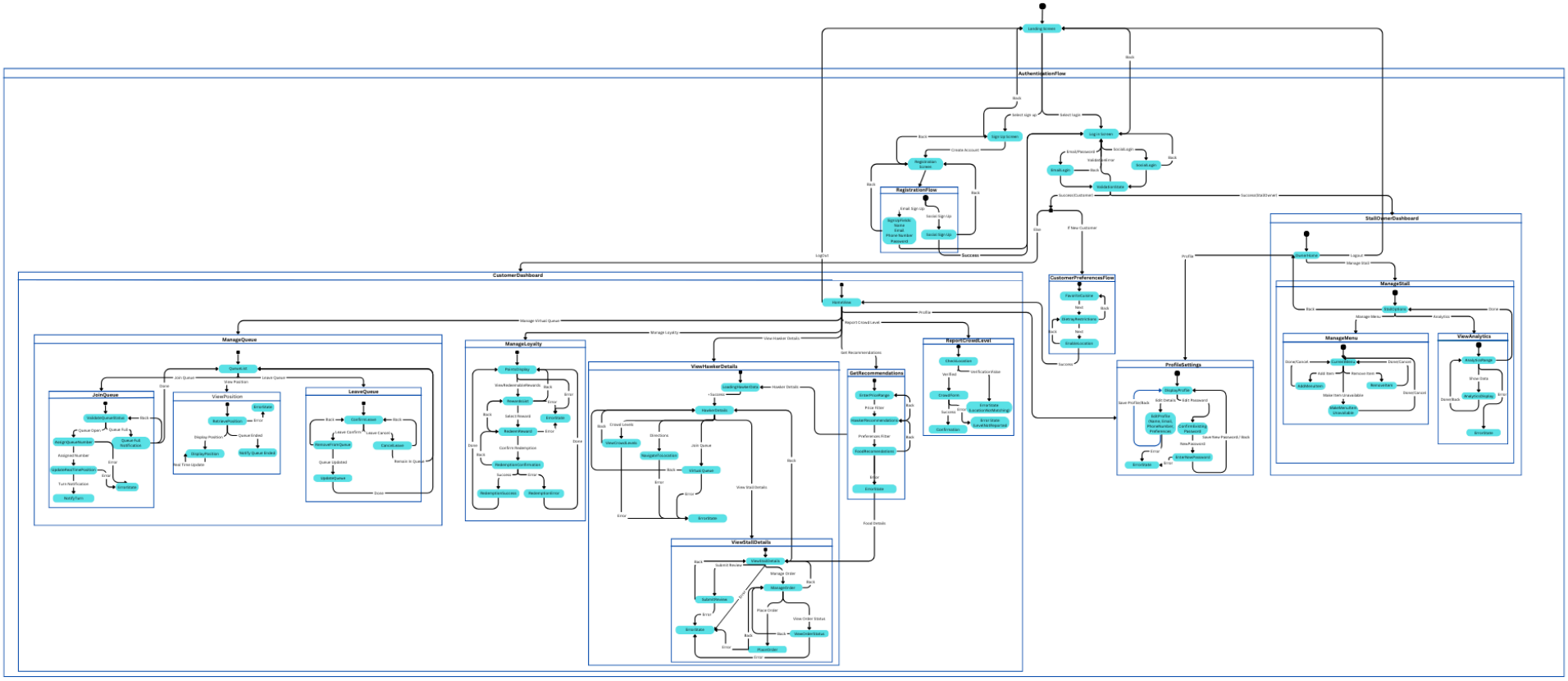
2.0.2 Manage Stall Menu



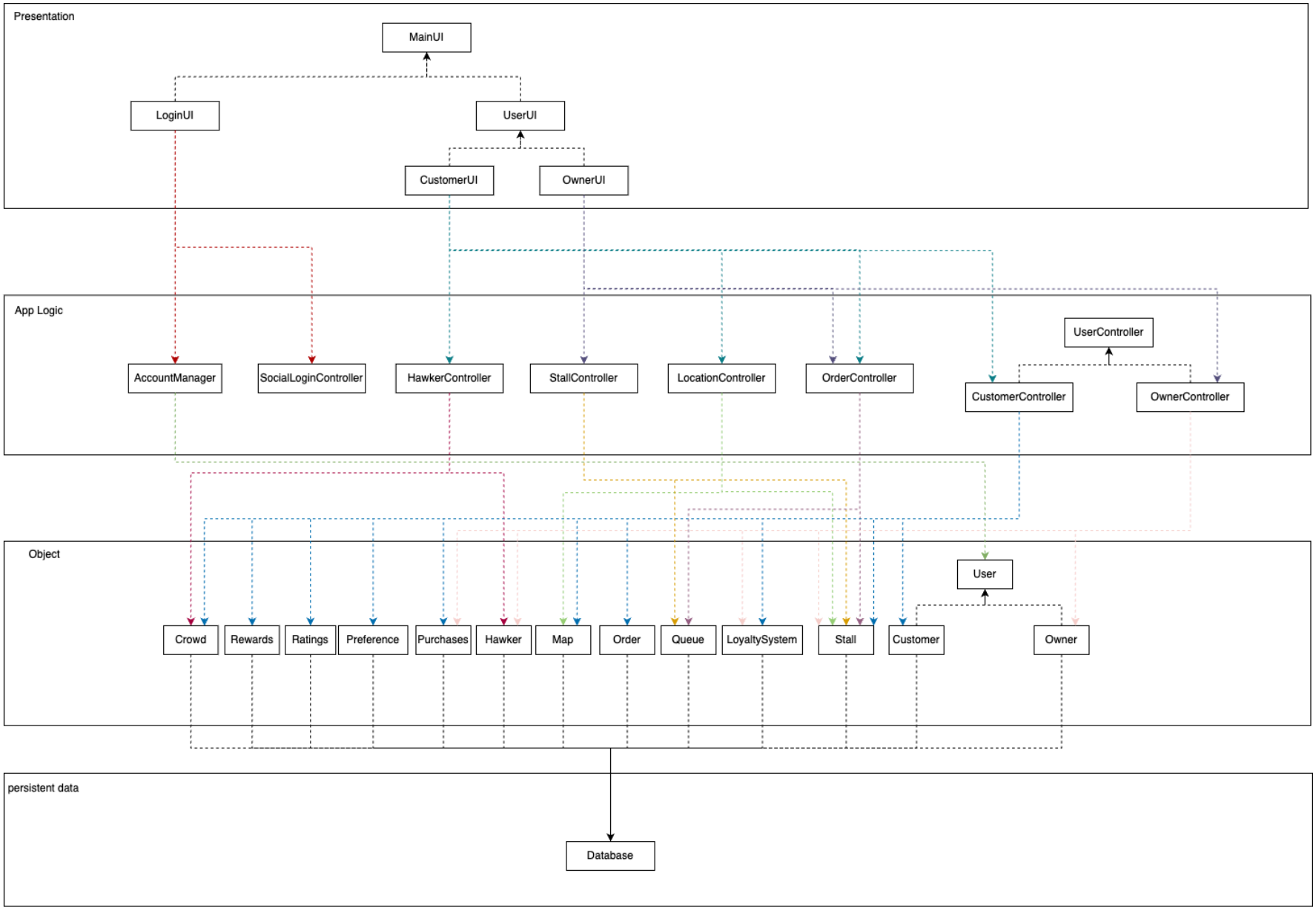
2.0.3 View Stall Metrics


Visual Paradigm Standard(Le Nguyen Bao Huy(Nanyang Technological University))
sd [2.0.3 View Stall Metrics]
StallOwner
OwnerUI
Stall Controller
Order Controller
1: viewStallMetrics
1.1: getStallMetrics
1.1.1: getAllOrders(stall_id)
1.1.2: ordersList
1.1.3: filterResolvedOrders
1.1.4: calculateMetrics
1.1.5: displayStallMetrics

6. Initial Dialog Map



7. System Architecture



Presentation Layer

This layer is responsible for Users to interact with HawkerGo. The different User Interfaces (UI) will call the corresponding controllers to implement the Application logic. This layer consists of:

1. **LoginUI**

   Allow users to log into the app through a username, email and password

2. **MainUI**

   Consists of LoginUI and UserUI

3. **UserUI**

   Consists of CustomerUI and OwnerUI

4. **CustomerUI**

   CustomerUI is part of UserUI, it allows customers to do their tasks via CustomerController

5. **OwnerUI**

   OwnerUI is part of UserUI, it allows Owners to do their tasks via OwnerController

App Logic layer

This layer contains all controller classes that will provide the presentation layer with their corresponding services. The controller classes will request for entities from the Object layer when it runs its logic. This layer consists of:

1. **AccountManager**

   AccountManager is called by LoginUI for Users to log in to HawkerGo. AccountManager includes the log in and sign up methods

2. **SocialLoginController**

   SocialLoginController is called by LoginUI for Users to log in to HawkerGo using existing social media accounts like Google and Facebook. It includes the get and set methods for both social logins

3. **HawkerController**

   HawkerController is called by CustomerUI to view the stalls in a Hawker Centre using the getAllStalls method

4. **StallController**

   StallController is called by OwnerUI to manage the details of their stalls by using the methods updateMenu, updateDescription etc.

5. **LocationController**

   LocationController is called by CustomerUI for Customers to get the location of a stall

6. **OrderController**

   OrderController is called by CustomerUI and OwnerUI for Customers and Owners to manage their orders using the methods, addOrder, removeOrder and pastOrders

7. **CustomerController**

   CustomerController is called by CustomerUI and contains the logic for Customer-specific interactions

8. **OwnerController**

   OwnerController is called by OwnerUI and it contains the logic for stall operations which includes menu and queue management and also viewing of sales metrics

App Object layer

This layer contains the entity classes that will be called by the App Logic Layer when it needs to run its logic. The entity classes will be stored in the database under the Persistent Data Layer. This layer consists of:

1. **Crowd**

   Crowd contains the crowd information like the current location, crowd level and hawker name

2. **Rewards**

   Rewards contains the rewards available, rewards claimed, stall name, hawker name and location

3. **Ratings**

   Ratings contain the hawker name, stall name, history and ratings

4. **Preference**

   Preference contains the cuisine, price lower bound and upper bound, whether it is halal/vegan or not, ranking and allergies

5. **Purchases**

   Purchases contains the stall name, food item, price and quantity

6. **Hawker**

   Hawker contains the stall name, profit, stall number, number of stalls and stall location

7. **Map**

   Map contains the current location, weather conditions and preferred location

8. **Order**

   Order contains the stall name, remark for kitchen, order ID, food item, whether it is halal/vegan, price and stall location

9. **Queue**

   Queue contains the queue number, order and status

10. **LoyaltySystem**

    LoyaltySystem contains the number of visits, stall name, location, hawker name and points

11. **Stall**

    Stall contains the food item, operating hours, allergies, whether it is vegan/halal, descriptions, spice level, price and availability

12. **Customer**

    Customer contains the location access

13. **Owner**

    Owner contains the description, stall number and stall name

Persistent Data Layer

This layer contains our database that will store all the entities

8. Application Skeleton

Client Frontend (React Native):

- assets: Contains images, fonts, and other static resources
- components: Reusable UI components organized by feature or type (buttons, cards, etc.)
- config: App configuration like API endpoints, environment variables
- constants: App-wide constants like colors, dimensions, text strings
- hooks: Custom React hooks for shared logic
- navigation: Navigation configuration using React Navigation
- screens: Screen components organized by feature (auth, profile, recommendations, etc.)
- services: API clients and services that interact with backend
- store: State management (using Redux or Context API)
- types: TypeScript type definitions and interfaces
- utils: Helper functions and utilities

Server Backend (Node.js/Express):

- config: Server configuration, environment variables, constants
- controllers: Route handlers organized by resource (users, hawkers, queues, etc.)
- middleware: Express middleware (auth, validation, error handling)
- models: Data models and schema definitions
- routes: API route definitions organized by resource
- services: Core business logic, external API integrations (LTA DataMall)
- utils: Helper functions, utilities, and common logic

Firebase:

- functions: Cloud functions for background tasks and triggers
- firestore-rules: Security rules for Firestore database

Documentation:

- Application setup guides, API documentation, architecture diagrams

HawkerGo Application Architecture

📱 Client (React Native)

Frontend mobile application for iOS and Android

```
app/                    Main application code
├── assets/             Static assets (images, fonts)
├── components/         Reusable UI components
├── config/             App configuration
├── constants/          Application constants
├── hooks/              Custom React hooks
├── navigation/         Navigation configuration
├── screens/            Screen components by feature
├── services/           API clients and services
├── store/              State management
├── types/              TypeScript type definitions
└── utils/              Helper functions
__tests__/              Frontend tests
App.js                  Application entry point
```

🖥️ Server (Node.js/Express)

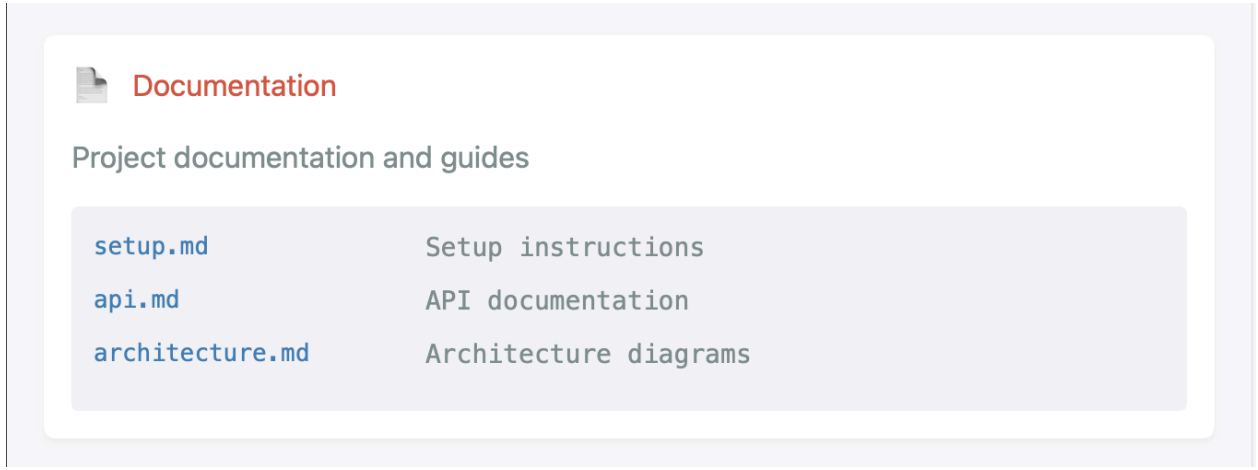
Backend API and business logic

```
src/                    Source code directory
├── config/             Server configuration
├── controllers/        Route controllers by resource
├── middleware/         Express middleware
├── models/             Data models
├── routes/             API route definitions
├── services/           Business logic, external APIs
├── utils/              Utility functions
└── app.js              Express application setup
__tests__/              Backend tests
server.js               Server entry point
```

🔥 Firebase

Database and cloud services

```
functions/              Cloud functions
firestore-rules/        Database security rules
```



9. Appendix

Links to Diagrams:

1. Use Case Diagram: https://drive.google.com/file/d/1b5mQbzj5zWwRN00mEU2g9IS1QcTHOp_2/view?usp=sharing
2. Key Classes Diagram: https://drive.google.com/file/d/1FrnirXQczm7AgvbfYqsjDugKznJyTSwN/view?usp=sharing
3. Entity, Boundary, Control Diagram: https://drive.google.com/file/d/1JeupuJWPHhK5jlhAlpkWr1ra0cfDI1HA/view?usp=sharing
4. Initial Dialog Map: https://www.canva.com/design/DAGf7-Q5IB8/nHQ913LXSbugzooTnukdoQ/view?utm_content=DAGf7-Q5IB8&utm_campaign=designshare&utm_medium=link2&utm_source=uniquelinks&utlId=h68ffb27ba6
5. System Architecture: https://drive.google.com/file/d/1XdJoX-xWO8e5pjtVFxLniejMfruL49gb/view?usp=sharing

Key Design Issues

A. Identifying and Storing Persistent Data

- Relational Database
    - User

        - **userid** (long), name (String), email (String), password (String, encrypted), user_type (String), google_login (String), facebook_login (String)
    - Map
        - **mapid** (String), userid (= userid), current_location (String)
    - Hawker
        - **hawkerid** (String), userid (= userid), hawker_name (String), profit (double), stall_number (str), num_of_stalls (int), mapid(= mapid)
    - Crowd
        - **crowdid** (int, AUTO_INCREMENT), userid (= userid), hawkerid (=hawkerid), crowd_level (String), record_timestamp (DATETIME, DEFAULT CURRENT_TIMESTAMP)
    - Stall
        - **stallid** (String), hawkerid (= hawkerid), userid (= userid), stall_name (String), allergies (String), vegan_friendly (bool), fooditem(String), spicelevel (String), halal (bool), price (double), operatinghour (String)
    - Rewards
        - **rewardsid** (String), userid (= userid), hawkerid (= hawkerid), stallid (= stallid), rewards_available (String), rewards_claimed (String)
    - Ratings
        - **ratingid** (String), userid (= userid), hawkerid (= hawkerid), stallid (= stallid), ratings (String), rating_history (String), record_timestamp (DATETIME, DEFAULT CURRENT_TIMESTAMP)
    - Preference
        - **preferenceid** (String), userid (= userid), preference (String)
    - Purchases
        - **purchaseid** (String), userid (= userid), hakwerid (= hawkerid), stallid (= stallid), foodPurchased (String), price (double), quantity (int)
    - Order
        - **orderid** (String), userid (= userid), purchaseid (= purchaseid)
    - Queue
        - **queueid** (String), userid (= userid), hawkerid (= hawkerid), stallid (= stallid), status (String), orderid (= orderid)
    - Loyalty_System
        - **loyaltyid** (String), userid (= userid), hawkerid (= hawkerid), stallid (= stallid), points (int), num_of_visits (int)

B. Providing access control

- **Encryption**
    - Encrypted password to be stored inside the database
- **Access Control**

Actors	**User**	**Map**	**Hawker**	**Crowd**	**Stall**	**Rewards**
User	createUser() updateUser() flagUser()	createMap() FlagMap() updateMap()	getHawker() getAllHawkers()	createCrowd() updateCrowd() getCrowd() flagCrowd()	getStall() getAllStalls()	getReward() useRewards() getAllRewards()
Stall Owner	createUser() updateUser() flagUser()	createMap() flagMap() updateMap()	createHawker() updateHawker() getHawker() getAllHawkers()	getCrowd()	createStall() updateStall() getStall() getAlllStalls()	createReward() updateReward() deleteReward() getReward() getAllRewards()

Actors	Ratings	Preference	Purchases	Order	Queue
User	createRating() updateRewards() getRating() getAllRatings() deleteRating() flagRating()	createPreference() updatePreference() deletePreference() getPreference()	createPurchase() getPurchase() getAllPurchases()	createOrder() getOrder() flagOrder() deleteOrder()	getQueue()
Stall Owner	getAllRatings() deleteRating() flagRating()	getPreference()	createPurchase() getPurchase() updatePurchase() deletePurchase() getAllPurchases()	getOrder()	createQueue() updateQueue() deleteQueue() getQueue()

C. Tech Stack

- Frontend
  - React Native
- Backend
  - Node.js/Express
- Database
  - Firebase