CSE6234 SOFTWARE DESIGN

FOOD DONATION AND REDISTRIBUTION SYSTEM

LECTURE SECTION: TC2L

TUTORIAL SECTION: TT4L

GROUP NAME: GROUP 4

GROUP MEMBERS:

	NAME	STUDENT ID
GROUP LEADER	MOHAMMED YOUSEF MOHAMMED ABDULKAREM	1221305727
GROUP MEMBER	MOHAMMED AAMENA MOHAMMED ABDULKAREM	1221305728
GROUP MEMBER	AMIRAH AISYAH BINTI AZMAN	1221305806
GROUP MEMBER	FARAH HANIM BINTI MOHD ZAMRI	1221305625

Table of Contents

Table of Figures

Table of Tables

1.0 Introduction

Food wastage and food insecurity are significant global issues that affect both developed and developing countries. According to the United Nations Environment Programme (UNEP), about 931 million tonnes of food are wasted annually across the supply chain (UNEP, 2021). In Malaysia, the situation is particularly concerning, with an estimated 17,000 tonnes of food being discarded every day, and over 24% of this food is still safe for consumption (SWCorp, 2022). At the same time, food insecurity continues to rise, especially among low-income communities in urban and rural areas, further exacerbated by the socioeconomic challenges brought on by the COVID-19 pandemic (Manaf et al., 2020). While there are some ongoing humanitarian efforts and unofficial donation campaigns, food redistribution in Malaysia remains inefficient, uncoordinated, and largely reactive. The lack of a central and intelligent platform that can automate the donation process, such as matching food requests, listing available surplus, and coordinating deliveries, makes it difficult to quickly and safely get food to those in need.

This project seeks to address this gap by creating a web-based Food Redistribution & Donation System, developed using the Django framework. The platform is designed to facilitate the entire food donation lifecycle, allowing food donors to list surplus items, while recipients can request, track, and receive food donations. Additionally, the system includes an administrative module to manage request approvals, monitor the platform's performance, and generate analytics reports.

The system is designed using five essential object-oriented design patterns to ensure it is modular, scalable, and easy to maintain. These design patterns include the Factory Pattern, which helps in creating user roles and encapsulating object creation; the Strategy Pattern, which allows for the dynamic selection and customization of food-matching algorithms based on different needs; and the Observer Pattern, which automatically updates food status, such as expiry dates or quantity changes. The Singleton Pattern is used to manage centralized application settings, ensuring there is only one source of truth for the system's configurations. Lastly, the Decorator Pattern provides flexibility in the notification system, enabling the use of multiple communication channels, such as email, SMS, or in-app alerts, without modifying the core notification logic. This platform not only addresses the critical issues of food waste and food insecurity but also aligns with Malaysia's Sustainable Development Goals (SDGs), specifically SDG 11 (Sustainable Cities and Communities) and SDG 12 (Sustainable Consumption and Production). By following global standards like ISO/IEC/IEEE 29148:2018 and adopting a user-centred design approach, the system offers a practical and scalable

solution that can be easily replicated and deployed in other developing countries, contributing to long-term social and environmental sustainability.

1.1 Abstract

In Malaysia, food waste and food insecurity are closely connected global issues. Every day, large amounts of edible food are thrown away, while many vulnerable groups continue to struggle with access to proper nutrition. Despite the efforts of charity organizations, the lack of a fully developed digital infrastructure makes it difficult to redistribute excess food quickly and efficiently to those in need. This project aims to address this issue by proposing a web-based Food Redistribution & Donation System built on the Django platform. The system is designed to offer a transparent, efficient, and scalable process for food donations, ensuring that surplus food can reach those who need it most in a timely manner.

The platform allows food donors to list surplus food, and recipients can view, request, and track food availability in real time. An administrative interface is also included for managing requests, validating users, handling notifications, and generating analytics reports. The system is designed with modern software engineering best practices to ensure it is modular, maintainable, and scalable, making it adaptable to future needs. To achieve these goals, the system uses five major object-oriented design patterns: Factory, Strategy, Observer, Decorator, and Singleton which helps structure the system to be reusable and organized. These patterns automate key processes, such as role-based user creation, food matching, notification delivery, and status tracking. By combining these strategies, the system creates a flexible and extensible framework that can grow with the needs of its users and adapt to complex situations.

Aligned with established software development standards, this initiative also supports the United Nations Sustainable Development Goals (SDGs), particularly SDG 11 (Sustainable Cities and Communities) and SDG 12 (Responsible Consumption and Production). In addition to providing a practical technical solution for a pressing social issue, the system offers a model for sustainable software development that can be repeated and applied in other emerging contexts, making it a valuable tool for future digital solutions in social good projects.

1.2 Problem Statement

Food wastage and food insecurity remain inadequately addressed due to a lack of scalable, technology-integrated solutions, despite their rising significance at both global and Malaysian levels. It is estimated that over 931 million tonnes of food are wasted every year globally, revealing a gaping mismatch between available food and requirement (UNEP, 2021). In Malaysia alone, 17,000 tonnes of food are wasted daily, with over 24% remaining edible (SWCorp, 2022).

Traditional food donation operations tend to rely heavily on traditional manual coordination, resulting in system inefficiencies, including inefficient traceability, delayed response time, and frequent mismatches between available supply and recipient demand. These are aggravated by the perishable quality of donated food, which calls for immediate distribution and rapid delivery to prevent spoilage and health risks (Lim et al., 2021).

Several critical limitations with existing systems have been identified:

1. The absence of automated matching algorithms considering urgency, food type, and geographic location.
2. A lack of centralized platforms capable of managing end-to-end donation processes, from listing to delivery.
3. Limited visibility and reporting mechanisms for administrators, donors, and policymakers to track impact and performance.
4. **Insufficient access control and role-specific functionality,** which undermines security, scalability, and usability across different user groups.

Addressing these challenges necessitates the development of a robust, modular, and role-based software solution. The system must be able to enable timely and secure redistributions of food resources while providing high levels of data integrity, scalability, and usability. The solution must also support real-time notifications, full reporting, and dynamic tracking of foods to improve transparency and operational efficiency across all stakeholders.

1.3 Project Objectives

The project's overall objective is to design and develop a scalable, web-based Food Redistribution and Donation System that minimizes food wastage and enhances food security among vulnerable groups. The system will ensure fair and efficient food redistribution based on intelligent automation, modular development, and simple accessibility.

The project objectives are enumerated as:

1. **To develop an online centralized platform** to enable listing, solicitation, monitoring, and reporting of food donations, and thus make the redistribution process transparent, traceable, and efficient.
2. **To possess secure role-based access control and user authentication mechanisms** for ensuring data integrity, operational accountability, and differentiated functionality for administrators, food donors, and recipients.
3. **To combine automatic food matching and real-time notification features with key software design patterns** in a way that donations on hand and recipients' requirements are matched dynamically based on urgency, food category, and location proximity and maintainability and scalability of the system are ensured.

1.4 Literature Review

1.4.1 Food Rescue US: A Volunteer-Based Food Recovery Platform

1.0 Introduction

Food Rescue US is an online system designed to eliminate hunger and food waste by bridging donors of food directly with social service agencies through volunteer drivers. Instead of relying on the traditional food banks, it instils a sense of community in rescuing and redistributing surplus food. This review considers how Food Rescue US operates, its key characteristics, its limitations, and the insights it offers towards the design of an efficient Food Redistribution and Donation System.

2.0 Context and Background

On a global level, 30–40% of food that is produced is lost or wasted, and this causes enormous waste of resources and pollution of the environment (FAO, 2022). In the United States alone, there are 108 billion pounds of lost food annually (Feeding America, 2023). Food Rescue US

emerged as an answer to inefficient food donation logistics within old systems, real-time recovery missions utilizing community volunteers and mobile devices, bridging the gap between surplus food and food insecurity within a region.

3.0 Technical Review of Food Rescue Platforms

3.1 Conceptual Overview

Food Rescue US operates as a mobile app and website where surplus food donors (restaurants, grocers, cafeterias) post available food. Registered volunteer drivers pick up volunteer driver assignments, which transport food directly to recipient organizations (shelters, food pantries).

3.2 Selected Existing System: Food Rescue US

The organization has expanded to more than 30 sites in the United States and employs a low-infrastructure model that maintains low costs and logistics overhead through the utilization of volunteer networks instead of paid staff.

Users can:

1. Post excess available with pickup time and location.
2. Take on-demand rescues through the app.
3. Track completed rescues and measure impact (e.g., pounds of food delivered).
4. Report rescue completion and delivery issues.

4.0 Key Features and Functionalities of Food Rescue Platforms

- **Mobile-First Interface**
  Preinstalled on mobile apps for donors and volunteers to ensure ease of coordination.
- **Real-Time Notifications**
  Volunteers receive instant notifications whenever there is a food rescue available in their region.
- **Impact Tracking**
  Volunteers and organizations can view the amount of food rescued and carbon emissions saved.
- **Scheduling Flexibility**
  Volunteers can choose rescues according to their schedule, thus providing a flexible engagement model.

- **Direct Rescue Model**
  Keeps warehouses out by shipping directly between donor and recipient, reducing the need for storage and food spoilage risk.

5.0 Challenges and Limitations

- **Volunteer Dependence**
  The model relies strongly on volunteer timetables, which occasionally fail to align with key donation times.
- **Quality Assurance Risks**
  As food comes into contact with volunteers, maintaining consistent safety and hygiene levels may prove challenging.
- **Limited Geographic Coverage**
  While successful in urban and suburban environments, rural areas face fewer volunteers and longer distances to cover.
- **Data Management and Integration**
  The framework has poor APIs to integrate with other food reporting or management systems, which restricts scalability.

6.0 Future Directions and Innovations

- **AI- Powered Scheduling**
  Predictive software can assign rescues to volunteers based on location, vehicle size, and time slots to optimize rescues.
- **Enhanced Food Safety Training**
  Adding compulsory brief training modules or certifications through the app can increase food handling quality.
- **Partnership Expansion**
  Partnership with corporate logistics fleets (ride-sharing services) can fuel last-mile delivery when volunteers are not present.
- **Blockchain-Based Traceability**
  Blockchain platforms would increase donor-to-recipient transparency and traceability, ensuring accountability and control of quality.

7.0 Summary and Critical Reflection

Food Rescue US advocates for an effective, community-based approach to anti-food waste efforts. It is strong where real-time matching, volunteer capacity, and low infrastructure expense exist. Scalability concerns, quality control, and rural community access remain high priorities to solve.

These outcomes will inform the design of the Targeted Food Redistribution and Donation System. In particular, dynamic matching procedures for donors-volunteers, food safety assurances, and real-time monitoring will be incorporated, which will maintain the platform scalable, secure, and efficient for future growth.

1.4.2 SeVa: A Food Donation App for Smart Living

1.0 Introduction

Loss of food has been an acute global issue, with a great impact on the environment, economy, and society. The growing necessity for sustainable measures has put digital technologies—mobile applications specifically—on the agenda as promising tools in reducing food surplus and hunger. With growing digitalization in the food industry, technology is increasingly applied to monitor, assess, and provide solutions for inefficiencies in the supply chain (Oroski & Silva, 2022). This literature review explains how mobile apps are involved in reducing food wastage via food donation, particularly in regions where food insecurity is prevalent.

2.0 Context and Background

At a worldwide level, as much as one-third of the total food that is produced gets wasted, contributing to a vast loss of resources and a greenhouse gas emitter (Chaudhari et al., 2021). Food wastage remains an issue of urgency in India as well, owing to logistical inefficiencies and infrastructure problems (Chaudhari et al., 2021). The co-existence of food waste and food insecurity highlights the need for efficient redistribution systems.
Food donation is more and more recognized as a viable strategy of wastage minimization while striving to alleviate hunger. It is the process of redistribution of surplus edible food to individuals and groups in need and consequently curtails environmental and humanitarian problems.

Technological innovations have enabled more efficient, scalable food redistribution mechanisms. Digital platforms, and particularly mobile applications, are being utilized as channels to enable food donation procedures, to improve donors and recipients' communication, and to enhance logistical coordination.

3.0 Technical Review of Mobile Food Donation Apps

3.1 Conceptual Overview

Mobile apps that are meant for food donation serve as online go-betweens, connecting food donors—who can be households, restaurants, and retailers—with recipient benefiting groups or institutions. The platforms aim to reduce food wastage by ensuring real-time coordination and simple interfaces.

3.2 Selected Existing System: SeVa

SeVa is a notable example of a mobile-based food donation app for surplus food management and hunger alleviation in India (Oroski & Silva, 2022). The app enables food donors to list available surplus while connecting efficiently to verified receiving organizations.

Other platforms, such as Annapurnata (Nalawade et al., 2024), provide the same food donation services but vary in scalability, functionality, and safety assurances.

4.0 Key Features and Functionalities of Food Donation Apps

- **User Interface and Usability**
  A simple to use and intelligent interface is paramount to facilitating active use by the donors and receivers. Accessibility and usability are integral to mass uptake.
- **Matching Algorithms**
  Efficient matching based on several parameters such as geospatial proximity, food quantity and type, and dietary specifications can be made using advanced algorithms.
- **Logistics and Coordination**
  Scheduling, pickup, and delivery functionalities are central to making food available for collection and redistribution. Mapping functionalities and notification functionalities enhance coordination.
- **Quality and Safety Mechanisms**
  Although most apps focus on redistribution, few possess robust quality control mechanisms. Testing food remains a significant weakness (Nalawade et al., 2024).
- **Information and Outreach**
  All websites offer information material, including food bank and shelter lists, food safety tips, and organizational contact information to enable repeated donation drives.

5.0 Challenges and Limitations

- **Accessibility Barriers**
  Despite the promise, digital solutions may not be accessible to all target users, particularly in communities where smartphone or internet access is limited (Nalawade et al., 2024).
- **Logistical Complexities**
  Transport and storage logistics are a critical issue. Obtaining food collected and delivered within safe time frames requires precise coordination.
- **Food Safety Issues**
  Hygiene and safety levels should be upheld throughout the donation process. Lacking regulation or built-in verification processes, the risk of contamination is higher.
- **Scalability Challenges**
  Scaling such platforms to function in larger areas or multiple cities necessitates heavy infrastructure and collaborative efforts of stakeholders, which might not always be possible.

6.0 Future Directions and Innovations

- **Technological Integration**
  The future might see the integration of food donation apps with new emerging technologies like IoT, blockchain, or AI to enhance monitoring, accountability, and efficiency.
- **Collaborative Ecosystems**
  Developer-non-profit food industry player-government collaboration will play an important role in filling structural barriers and ensuring sustained influence.
- **Filling Quality Assurance Gaps**
  Utilization of food quality assurance methods such as electronic temperature checks or expiration date tracking might help close the gaps available for exploitation for food safety purposes (Nalawade et al., 2024).

7.0 Summary and Critical Reflection

Smart phone applications are equally of high promise in helping prevent food loss and increase food safety. Although products like SeVa exemplify the promise of technological intervention, a long way to go remains as far as access, food safety, and matters of scalability concerns are concerned. This literature review attempts to find a user-oriented, modular, and

extensible model of food redistribution that offers dependable safety features, real-time logistics coordination, and advanced food matching algorithms. These findings feed into the current design choices in this project's Food Redistribution and Donation System to some immediate extent, taking leverage of software engineering best practice and object-oriented design patterns in a bid to leverage the recognized vulnerabilities of current solutions.

1.4.3 OLIO: A Community Food Sharing Application

1.0 Introduction

The increasing issues of food wastage and food insecurity have driven the exploration of community-level digital interventions. A key example of a community food sharing application is OLIO, a peer-to-peer (P2P) platform that facilitates the redistribution of surplus food among individuals and businesses. By forming hyperlocal sharing networks, OLIO seeks to prevent edible food from being wasted while also creating more connected communities. This literature review explores OLIO's operational model, functionalities, problems, and its relevance to food redistribution system design.

2.0 Context and Background

International food loss accounts for approximately 8–10% of total greenhouse gas emissions, hastening environmental degradation and accelerating resource depletion (UNEP, 2021). Within urban and suburban populations, the majority of waste occurs at the retail and consumer levels via over-purchasing, cosmetic standards, and inefficient distribution. Observing that a lot of food is being wasted even at the household and small business level, OLIO was developed to enable households, shops, and food service businesses to share surplus food in their areas, thus bridging surplus and need in the community.

3.0 Technical Review of Food Sharing Applications

3.1 Conceptual Overview

OLIO is a peer-to-peer (P2P) website where users create listings for surplus food items, and nearby users can view and request available items for free. The site is centred on hyperlocal sharing, real-time communication, and low logistical complexity by enabling users to coordinate pickups between themselves. OLIO aims primarily at households but has also expanded to work together with businesses and supermarkets for bulk food rescue operations.

3.2 Selected Existing System: OLIO

OLIO was established in 2015 in the United Kingdom and has since expanded to the international market. The application allows users to:

1. List surplus food with descriptions, images, and pickup details.
2. Search local available food options by location.
3. Chat directly using built-in messaging features.
4. Organize pickups by mutual agreement between donor and recipient.
5. Join volunteer programs like "Food Waste Heroes," where volunteers go around collecting unsold food from retailers for redistribution (OLIO, 2023).

4.0 Key Features and Functionalities of Food Sharing Applications

- **User Interface and Usability**
  OLIO possesses a straightforward, easy-to-use interface with geolocation-based product searching, push alerts, and messaging integration to make sharing easy.
- **Geolocation and Hyperlocal Matching**
  Proximity listings are given preference to facilitate on-time pickup and minimize transportation emissions.
- **Volunteer Integration**
  "Food Waste Heroes" amplify OLIO's reach by working with supermarkets and recovering surplus food for redistribution, bridging individual and institutional food donation.
- **Communication Tools**
  Integrated chat features allow real-time negotiation of pickup details, enhancing responsiveness and adaptability.
- **Expansion Beyond Food**
  OLIO also allows sharing of non-food products like toiletries and household goods, increasing the scope for sustainability.

5.0 Challenges and Limitations

- **Trust and Safety Concerns**
  As a peer-to-peer initiative, OLIO relies on the reliability of users regarding food quality and freshness, without systematic check-verification.

- **Limited Access for Technologically Underserved Users**
  Participation is conditional upon the presence of smartphones and consistent internet connectivity and hence can potentially bar poor segments.
- **Pickup Coordination Complexity**
  Pickup is based on bilateral planning, and this can result in lateness or cancelled appointments.
- **Scale Limitations in Rural Areas**
  The OLIO model works optimally in densely populated urban cities with more dense users; performance is low under low-density scenarios.

6.0 Future Directions and Innovations

- **Food Quality Assurance Mechanisms**
  Merging user rating feedback and potential third-party certification of donated food could enhance safety and trust.
- **AI-Based Matching Algorithms**
  Future releases could leverage AI to predict best matches between donors and recipients by food type, degree of urgency, and pickup probability likelihood.
- **Greater Business Partnerships**
  More partnership with larger chain stores and bulk collection logistics integration could reach beyond direct users for OLIO.
- **Multilingual and Accessibility Features**
  Making the apps translated and available to the disabled would make the platform more accessible.

7.0 Summary and Critical Reflection

OLIO is a new fight against food waste with community-centred technology. Its hyperlocal model of sharing supports efficient sharing of surplus food, enhancing local connectivity, and reducing environmental expenses. Food quality assurance, accessibility, and coordination logistics challenges must be addressed to gain maximum potential.

Learning from OLIO's operational success and limitations is brought to bear in designing the proposed Food Redistribution and Donation System with emphasis on trust mechanisms, real-time logistics control, and inclusive design principles to create a scalable and resilient solution.

1.4.4 Literature Review Summary

Table 1: 1.4.4.1 – Literature Review Summary

Feature	Food Rescue US (USA)	SeVa (India)	OLIO (Global)
Primary Model	Volunteer-driven direct rescue (donor → agency)	Mobile app-based donor-recipient matching	Peer-to-peer (P2P) hyperlocal sharing
Key Users	Restaurants, grocers, volunteers, nonprofits	Households, restaurants, NGOs	Individuals, businesses, volunteers
Core Features	- Real-time rescue alerts - Impact tracking (lbs saved) - No warehousing	- Matching algorithms - Logistics coordination - Food safety tips	- Geolocation-based listings - In-app messaging - Non-food item sharing
Strengths	- Low infrastructure - Scalable in urban areas -Real-time coordination	- User-friendly interface - Focus on food safety education	- Community-building - Flexibility (food + non-food) - Volunteer programs (e.g., "Food Waste Heroes")
Challenges	- Volunteer dependency - Limited rural coverage - Food safety risks	- Digital divide (accessibility) - Logistical complexities - Scalability issues	- Trust/safety concerns - Tech barriers for low-income users - Rural scalability
Future Improvements	- AI-powered scheduling - Blockchain traceability - Corporate logistics partnerships	- IoT/blockchain integration - Govt-NGO partnerships - Enhanced QA mechanisms	- AI-based matching - Business partnerships - Multilingual/accessibility features
Relevance to FRDS	Dynamic donor-volunteer matching, real-time tracking	Modular design, safety protocols	Hyperlocal focus, trust mechanisms

1.5 Project Scope

The goal of this project is the design, development, and deployment of a web-based food redistribution and donation system that serves as a conduit for the effective transfer of surplus food from donor to recipient and hence, minimization of food wastages while maximizing access to food for those communities that can barely afford it. The system enables food donors such as individuals, restaurants, and organizations to easily reach the required recipients, comprised of needy individuals and NGOs, through a highly intelligent and structured platform.

This entire project will consist of the entire software development lifecycle from requirement analysis to system design and implementation using the Django framework, testing, and finally deployment augmented by documentation. It as well covers the design of necessary schematics (ERD, Use Case, Class) and the entire application of software engineering principles and design patterns in ensuring code modularity, scalability, and maintenance.

1.6 System Scope

The Food Redistribution and Donation System is a comprehensive digital platform that manages the entire lifecycle of food donation. It has three relieving user roles such as Food Donor, Food Recipient, and Admin. Each of these user roles bears specific responsibilities and access privileges.

Core Functionalities:

- **User Management**: Registration and login, role-based access control, and profile management are secured.
- **Donation management**: Donors are allowed to list, update, and manage items available for donation.
- **Request System**: Browsing of food available by recipients and requesting is done alongside checking the status.
- **Pickup Coordination**: The donor and recipient will manage and schedule pickups and can confirm pick up by the recipient.
- **Notification**: System triggered events will generate real-time updates trigger notifications such as not limited to approvals and reminders.
- **Feedback Mechanism**: Feedback from users is collected after donation transactions.

- **Administrative Tools**: Admin controls the user admin for donations and requests of amount and generates reports and configure rules.
- **Inventory and Transaction Tracking**: It deals with quantities of available foods and records transaction history.

The system operates through a web-based interface accessible from desktop browsers. Nevertheless, it does not include offline donations and does not employ third-party logistics integration.

1.6.1 Food Donor Use Cases

1.6.1.1 Register Account

Feature	**Register Account**
Version	1.0
Purpose	To allow new food donors to create an account on the system.
Pre-condition	The user has not registered before.
Post-Condition	The user has a registered account and can log in.
Main Flow	1. The user navigates to donor registration page. 2. The system displays registration form. 3. User fills in personal details (name, email, password, etc.) 4. User submits the form. 5. System validates input and creates the account. 6. Confirmation message is displayed.
Alternate Scenario	• If the email provided already exists in the database, an error message is shown. • If any field is missing or invalid, the system prompts the donor to correct the input.

1.6.1.2 Log In

Feature	**Log In**
Version	1.0
Purpose	To allow new food donors to authenticate and access the system.
Precondition	The user has a valid registered account.
Postcondition	The user is logged in and redirected to their dashboard.
Main Flow	1. The user navigates to the food donor login page.

	2. The system displays the login form. 3. The user enters username and password. 4. User submits the form. 5. System validates credentials. 6. System logs in the user and redirects them.
Alternate Scenario	• If credentials are incorrect, error message is displayed. • If fields are empty, user is prompted to complete them.

1.6.1.3 Add Food Listing

Feature	**Add Food Listing**
Version	1.0
Purpose	To allow food donors to create a new listing for surplus food available for donation.
Precondition	User is logged in and has a valid donor account.
Postcondition	A new food listing is created and added to the platform's inventory, visible to recipients.
Main Flow	1. The user logs into the platform. 2. User navigates to the "Add Food Listing" section. 3. User fills in the required details (e.g., food name, quantity, expiration date, pickup location). 4. User submits the form. 5. System validates the input. 6. If all inputs are valid, system saves the new listing into the database. 7. System confirms successful listing creation and displays it on the donor's dashboard.
Alternate Scenario	• If required fields are left empty, the system prompts the user to complete them before submission. • If input values are invalid (e.g., negative quantity, past expiration date), the system rejects the form and highlights errors. • If the server/database connection fails, an error message is displayed, and the submission is retried.

1.6.1.4 Edit Food Listing

Feature	**Edit Food Listing**
Version	1.0
Purpose	To allow food donors to modify details of a listed food item.
Precondition	The user is logged in and has listed food items.
Postcondition	The updated food listing is saved in the system.
Main Flow	1. User navigates to their food listings. 2. User selects a listing to edit. 3. System displays current listing details. 4. User modifies desired details. 5. User submits changes. 6. System saves and confirms update.
Alternate Scenario	• If input is invalid, error message is shown. • If listing no longer exists, user is notified.

1.6.1.5 Delete Food Listing

Feature	**Delete Food Listing**
Version	1.0
Purpose	To allow food donors to remove a listed food item from the platform.
Precondition	The user is logged in and owns the listing.
Postcondition	The selected food listing is removed from the database.
Main Flow	1. User navigates to their food listings. 2. User selects an item to delete. 3. System prompts for confirmation. 4. User confirms. 5. System deletes the listing and notifies the user.
Alternate Scenario	• If deletion fails, system notifies the user.

1.6.1.6 View Donation History

Feature	View Donation History
Version	1.0
Purpose	To allow food donors to view a log of their past donations.
Precondition	User is logged in.
Postcondition	User views donation history records.
Main Flow	1. User navigates to Donation History. 2. System retrieves and displays historical donation data.
Alternate Scenario	• If no donation record exists, system displays an empty state message.

1.6.1.7 Update Profile

Feature	Update Profile
Version	1.0
Purpose	To allow users to edit their personal and contact information.
Precondition	User is logged in.
Postcondition	Updated profile details are stored.
Main Flow	1. User navigates to profile settings. 2. System displays current profile details. 3. User edits field and submits. 4. System saves updates and confirms.
Alternate Scenario	• System flags invalid input.

1.6.1.8 Manage Pickup Schedules

Feature	Manage Pickup Schedules
Version	1.0
Purpose	To allow food donors to set or update pickup time and location.
Precondition	User is logged in and has at least one active listing.
Postcondition	Pickup information is updated in the system.
Main Flow	1. User accesses Pickup Schedule section. 2. System displays current schedules. 3. User modifies time and location. 4. System saves changes.

Alternate Scenario	• Conflicts or invalid times prompt an error message.

1.6.1.9 Submit Feedback

Feature	**Submit Feedback**
Version	1.0
Purpose	To allow users to give feedback on their experience.
Precondition	User is logged in.
Postcondition	Feedback is submitted and stored.
Main Flow	1. User navigates to Feedback section. 2. User writes and submits feedback. 3. System stores feedback and displays confirmation.
Alternate Scenario	• Missing required fields prompt validation message.

1.6.2 Food Recipient Use Cases

1.6.2.1 Register Account

Feature	**Registration Account**
Version	1.0
Purpose	To allow a food recipient to create an account by submitting their necessary details such as name, email, contact number, password, and location.
Precondition	• The system is operational and accessible. • The recipient has valid details for registration and does not register before.
Postcondition	• The recipient is successfully registered, and an account is created in the system. • The recipient able to log in and perform subsequent actions.
Main Flow	1. The recipient navigates to recipient registration page. 2. The system prompts the recipient to enter their name, email address, contact number, password, and address. 3. The recipient submits the registration form. 4. The system validates the input.

	5. If the credentials are correct. The system creates the recipient account and stores it in the database. 6. Confirmation message is displayed.
Alternate Scenario	• If any field is missing or invalid, the system prompts to correct the respective input. If the email provided already exists in the database, the system prompts the recipient that the email already exists.

1.6.2.2 Log In

Feature	**Log In**
Version	1.0
Purpose	This use case allows a recipient to log in to the system using their credentials.
Precondition	• The system is operational and accessible. • The recipient is registered in the system.
Postcondition	• The recipient is successfully logged in.
Main Flow	1. The recipient navigates to the login page. 2. The system prompts for email and password. 3. The recipient enters their credentials. 4. The system validates the credentials and redirects the recipient to their dashboard.
Alternate Scenario	• If the entered username or password is incorrect, the system returns an error “Invalid username or password". • If the user does not have an account, they can click the “Register” option to navigate to the recipient registration page.

1.6.2.3 Update Profile

Feature	**Update Profile**
Version	1.0
Purpose	This use case allows recipients to modify their profile information, including personal details and contact preferences.
Precondition	• The system is operational and accessible. • The recipient has been logged in.
Postcondition	The recipient’s profile is updated in the system.

Main Flow	1. The recipient accesses the profile management section in their dashboard. 2. The system displays current profile details. 3. The recipient edits the necessary fields. 4. The system validates the input. 5. The system updates the recipient profile in database. 6. A confirmation message is displayed.
Alternate Scenario	• If input validation fails, the system highlights errors fields and requests corrections.

1.6.2.4 Browse Available Food

Feature	**Browse Available Food**
Version	1.0
Purpose	To allow food recipient to easily find and select suitable food donations by browsing and filtering available listings based on category, expire, location.
Precondition	• The system is operational and accessible. • The recipient is authenticated and logged into their account.
Postcondition	• The receipt successfully views and interacts with a filtered list of available food item by default or based on the selected strategy. • The recipient can make a food request by click on “Request” button.
Main Flow	1. The recipient selects "Browse Food" from the dashboard. 2. The system retrieves all active food listings from the database. 3. The system displays all food items available. 4. The recipient applies the desired filters. 5. The system displays food items with filters (e.g., category, expiry, location). 6. The recipient reviews item details and availability. 7. The recipient may proceed to request an item by clicking the “Request” button.
Alternate Scenario	• If no food listings are available in the system. The system displays the message: “No food items are currently available. Please check back later”

	• If the recipient does not select any sorting preference. The system applies the default expire strategy.

1.6.2.5 Submit Food Requests

Feature	**Submit Food Request**
Version	1.0
Purpose	This use case enables recipients to request food items that are listed as available.
Precondition	• The recipient is logged in. • At least one food listing available.
Postcondition	• The food request is submitted and stored in the databased. • Pending to donor for pickup coordinate.
Main Flow	1. The recipient browses food listings. 2. The recipient selects an item and clicks "Request" on selected item. 3. The system prompts for pickup preferences and instructions. 4. The recipient submits the request. 5. The system records the request and notifies the donor.
Alternate Scenario	• If the item becomes unavailable during the process, the system prevent recipient to complete the request process.

1.6.2.6 Track Request Status

Feature	**Track Request Status**
Version	1.0
Purpose	This use case allows recipients to monitor the status of their food requests.
Precondition	• The recipient is logged in. • At least one food request exists.
Postcondition	• The recipient stays informed about their request progress.
Main Flow	1. The recipient navigates to the "My Requests" section. 2. The system lists all submitted requests and their statuses (e.g., Pending, Approved, Rejected, Completed). 3. The recipient reviews each status.

Alternate Scenario	• If no requests exist, the system shows “No active requests found.”

1.6.2.7 Cancel Food Request

Feature	**Cancel Food Request**
Version	1.0
Purpose	This use case allows recipients to cancel a food request before it is picked up.
Precondition	• The recipient is logged in. • A pending or approved request exists.
Postcondition	• The food request is removed from active requests. • The donor is informed.
Main Flow	1. The recipient navigates to the "My Requests" section. 2. The system displays active requests. 3. The recipient selects a request and clicks "Cancel." 4. The system asks for confirmation. 5. Upon confirmation, the request is marked as cancelled. 6. The donor is notified about the cancellation.
Alternate Scenario	• If the request is already marked as "Picked Up," cancellation is reject.

1.6.2.8 Manage Pickup Schedule

Feature	**Manage Pickup Schedule**
Version	1.0
Purpose	This use case allows a recipient to schedule or update a food pickup appointment with a donor.
Precondition	• The system is operational and accessible. • The recipient had been logged in the system. • A confirmed or accepted food request exists.
Postcondition	• The pickup schedule is updated in the system. • Both donor and recipient are notified.
Main Flow	1. The recipient navigates to the "Pickup Schedule" section. 2. The system displays available time slots and pickup details. 3. The recipient selects a preferred time and confirms it.

	4. The system updates the pickup schedule and notifies the donor. 5. A confirmation message is displayed.
Alternate Scenario	• If the time slot is not available, the system prompts the recipient to select another time or check back later. • If the donor reschedules, the system notifies the recipient to reselect the preferred time and confirms it.

1.6.2.9 Confirm Pickup

Feature	**Confirm Pickup**
Version	1.0
Purpose	To enable a food recipient to confirm that the donated food item has been successfully received from the donor.
Precondition	• The food recipient is logged into the system. • The recipient has an active food request. • The pickup approved and date have been scheduled.
Postcondition	• The system marks the food request as "Picked Up" in the database. • The pickup status is updated for both the donor and recipient.
Main Flow	1. The food recipient navigates to the "My Requests" section. 2. The recipient selects a request with status " Pending". 3. The recipient clicks the "Confirm Pickup" button. 4. The system updates the status to "Picked Up". 5. A success message is displayed, and a confirmation notification is sent to the donor.
Alternate Scenario	• If the recipient attempts to confirm before the scheduled pickup time, the system displays an error: “Pickup cannot be confirmed before the scheduled time.” • If the request status is already marked as "Picked Up" or "Cancelled", the confirm option is disabled.

1.6.2.10 Submit Feedback

Feature	**Submit Feedback**
Version	1.0
Purpose	This use case allows recipients to submit feedback about their donation experience.

Precondition	• The recipient has completed at least one food pickup. • The recipient is logged in.
Postcondition	• Feedback is stored and linked to the recipient's account.
Main Flow	• The recipient navigates to the feedback section. • The system displays a feedback form. • The recipient provides a rating and comments. • The recipient submits the form. • The system stores the feedback for analysis purposes and displays a confirmation message.
Alternate Scenario	• If required fields are missing, the system prompts the user to complete them before submission.

1.6.3 Administrator Use Cases

1.6.3.1 Log In

Feature	**Log In**
Version	1.0
Purpose	To allow Admins to authenticate and securely access the administration dashboard.
Precondition	Admin must have a valid registered account.
Postcondition	Admin is authenticated and redirected to the Admin dashboard.
Main Flow	1. Admin navigates to the login page. 2. System displays the login form. 3. Admin enters username and password. 4. Admin submits the form. 5. System validates credentials. 6. Admin is granted access to the dashboard.
Alternate Scenario	• If credentials are incorrect, system shows an error. • If fields are missing, system prompts to complete them.

1.6.3.2 Manage Requests and Transactions

Feature	**Manage Requests and Transactions**
Version	1.0
Purpose	To allow Admins to view, approve, reject, monitor all food requests and their associated transactions.

Precondition	Admin is logged in.
Postcondition	Transaction and request statuses are updated accordingly.
Main Flow	1. Admin navigates to the transactions and requests section. 2. System displays all the submitted requests and their statuses. 3. Admin reviews pending requests. 4. Admin approves or rejects requests. 5. System updates and notifies the relevant users.
Alternate Scenario	• If no requests exist, system shows an empty list. • If approval fails, an error is logged.

1.6.3.3 Moderate Food Listings

Feature	**Moderate Food Listings**
Version	1.0
Purpose	To allow Admins to review and remove inappropriate or expired food listings.
Precondition	Admin is logged in.
Postcondition	Listings are updated or removed as necessary.
Main Flow	1. Admin navigates to the Food Listing section. 2. System displays all active listings. 3. Admin reviews and flags inappropriate listings. 4. Admin deletes or updates flagged listings. 5. System confirms action and updates the list.
Alternate Scenario	• If no listings exist, system shows an empty message.

1.6.3.4 Manage Accounts

Feature	**Manage Accounts**
Version	1.0
Purpose	To allow Admins to view, edit, activate, deactivate, or delete user accounts.
Precondition	Admin is logged in.
Postcondition	User account data is updated or modified.
Main Flow	1. Admin navigates to the User Management section. 2. System displays the lists of all users. 3. Admin selects a user to view, edit, deactivate, or delete.

	4. Admin applies changes. 5. System saves updates and displays confirmation.
Alternate Scenario	• If update fails, system shows an error.

1.6.3.5 Monitor Pickups

Feature	**Monitor Pickups**
Version	1.0
Purpose	To allow Admins to monitor scheduled pickups and their status.
Precondition	Admin is logged in.
Postcondition	Pickup status is monitored, and discrepancies are reported.
Main Flow	1. Admin navigates to the Pickup Monitoring section. 2. System displays scheduled pickups and their statuses. 3. Admin reviews and identifies missed or delayed pickups. 4. Admin flags issue for investigation.
Alternate Scenario	• If no pickups are scheduled, system shows empty state.

1.6.3.6 Generate Reports

Feature	**Generate Reports**
Version	1.0
Purpose	To allow Admins to generate summary reports on system activities like donations, requests, and pickups.
Precondition	Admin is logged in.
Postcondition	Report is generated and available for download.
Main Flow	1. Admin navigates to the Reports section. 2. System displays reporting options (date range, type). 3. Admin selects criteria and requests report generation. 4. System processes and generates report. 5. Admin downloads or views the report.
Alternate Scenario	• If no data matches the selected criteria, system informs Admin and does not generate a report.

2.0 System Overview

2.1 Generic Use Case

A Use Case Diagram for the Food Redistribution and Donation System delineates the very interactions that happen among three major actors-Food Donor, Food Recipient, and Admin-himself among the listed functions. According to the system use cases, Food Donors will be able to manage food listings, view donation history, and update pickup details; Food Receivers can find food availability, submit and track requests, manage pickups, and perform confirmation on receipt of food. On the other hand, Administrators manage the whole workings of the platform, including user management, request management, and report generation. Important relationships such as <<include>> and <<extend>> denote logical dependencies of use cases—for example, submitting feedback is included after a confirmed pickup. This becomes another way to ensure modularity and scalability of the system and maintain a separation of concerns according to the responsibilities of each actor.

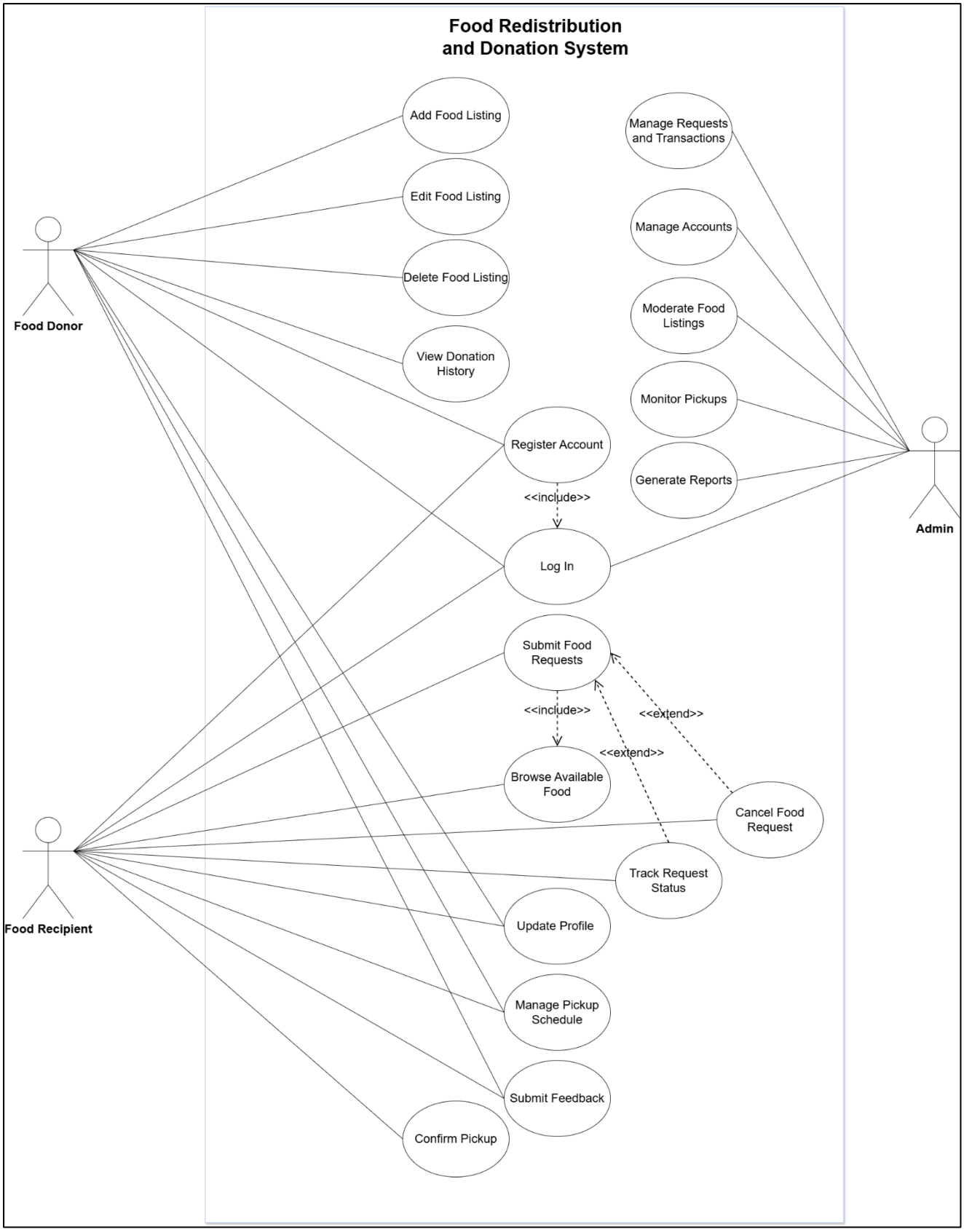

Figure 1: 2.1.1 – Use Case Diagram

2.2 Class Diagrams

Food Redistribution and Donation System UML Class Diagram is one of the representations of static system structure through classes, attributes, methods, and their relationship. Major classes employed in this representation are User, FoodDonor, FoodRecipient, Admin, FoodItem, Request, RequestItem, Transaction, Inventory, Feedback, Notification, and Report. They are each encapsulating some of the responsibility to the functional requirement of the system.

The Class Diagram for Food Redistribution and Donation System illustrates the basic organization and relationships of the major system components. Fundamentally, the system employs generalization in defining commonalities by a User class from which Admin, FoodDonor, and FoodRecipient are specialized sub-classes. Admins manage users and reporting while FoodDonors add food and FoodRecipients place requests for food items in availability. A Request aggregates a number of RequestItems, and each Request is associated with one or more FoodItems, which form the groundwork for transactions. Transactions track pickup status and levels of food donations, and Inventory oversees quantity of products and pickup time. The system also supports user interaction through Feedback and Notification classes to enable real-time communication and continuous system refinement. The diagram follows object-oriented paradigms of modularity, encapsulation, and single responsibility, and is indicative of a scalable as well as sustainable solution.

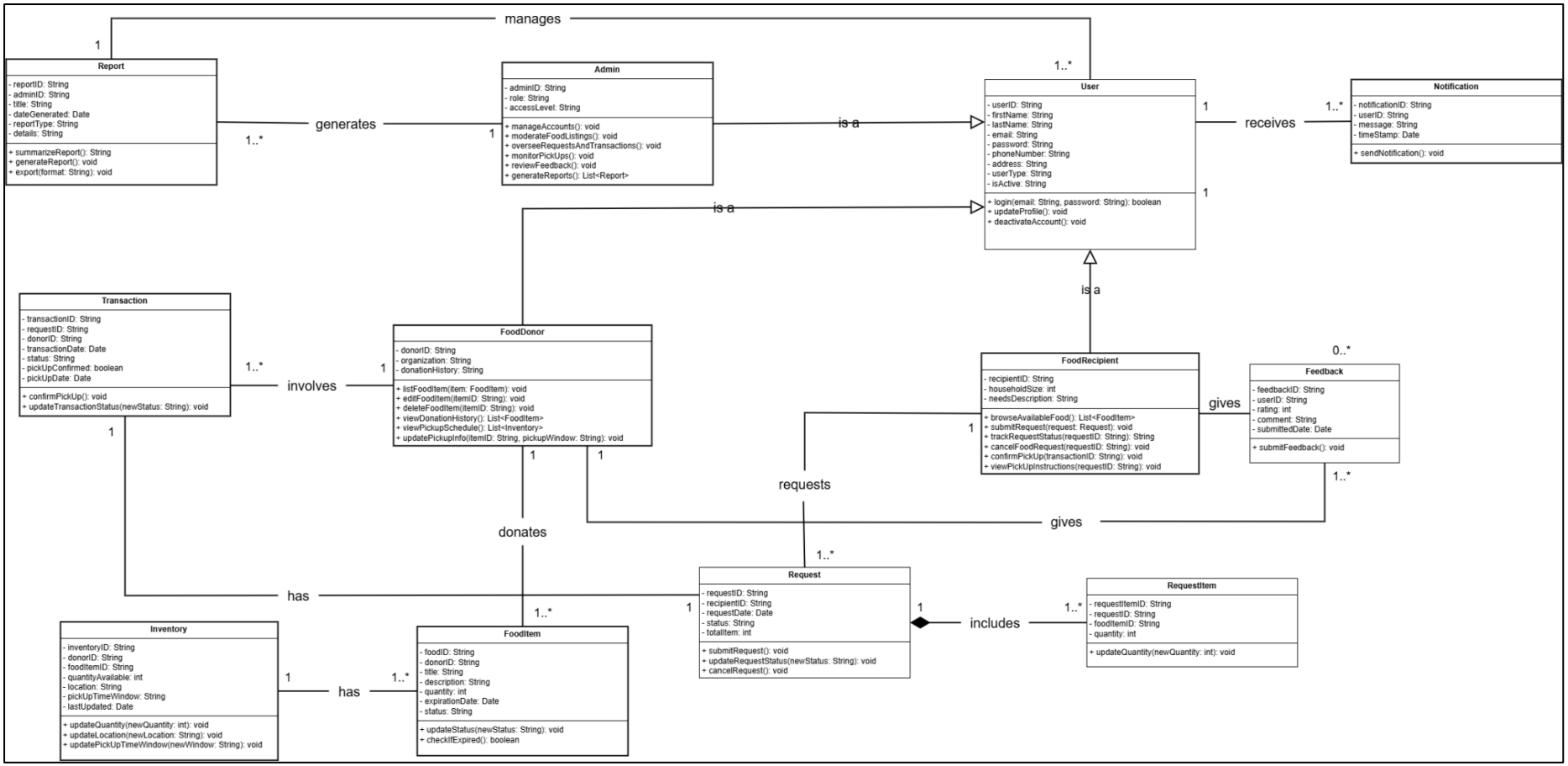

Figure 2: 2.2.1 – Food Donation and Redistribution Class Diagram

2.3 Entity Relationship Model

The Entity-Relationship Diagram (ERD) of the Food Redistribution and Donation System identifies the key entities, attributes, and relationships to keep record of food donations, requests, transactions, and user activities. The entities User, FoodDonor, FoodRecipient, FoodItem, Admin, Request, RequestItem, Transaction, Inventory, Notification, Report and Feedback are linked through well-designed primary and foreign keys for ensuring data consistency.

The ERD captures the most important relationships, including donation flow between recipient and donor, inventory control, and administration control. Organizing the system database in an organized manner, the ERD provides a strong foundation for ensuring data consistency, scalability of the system, and efficient performance of the platform.

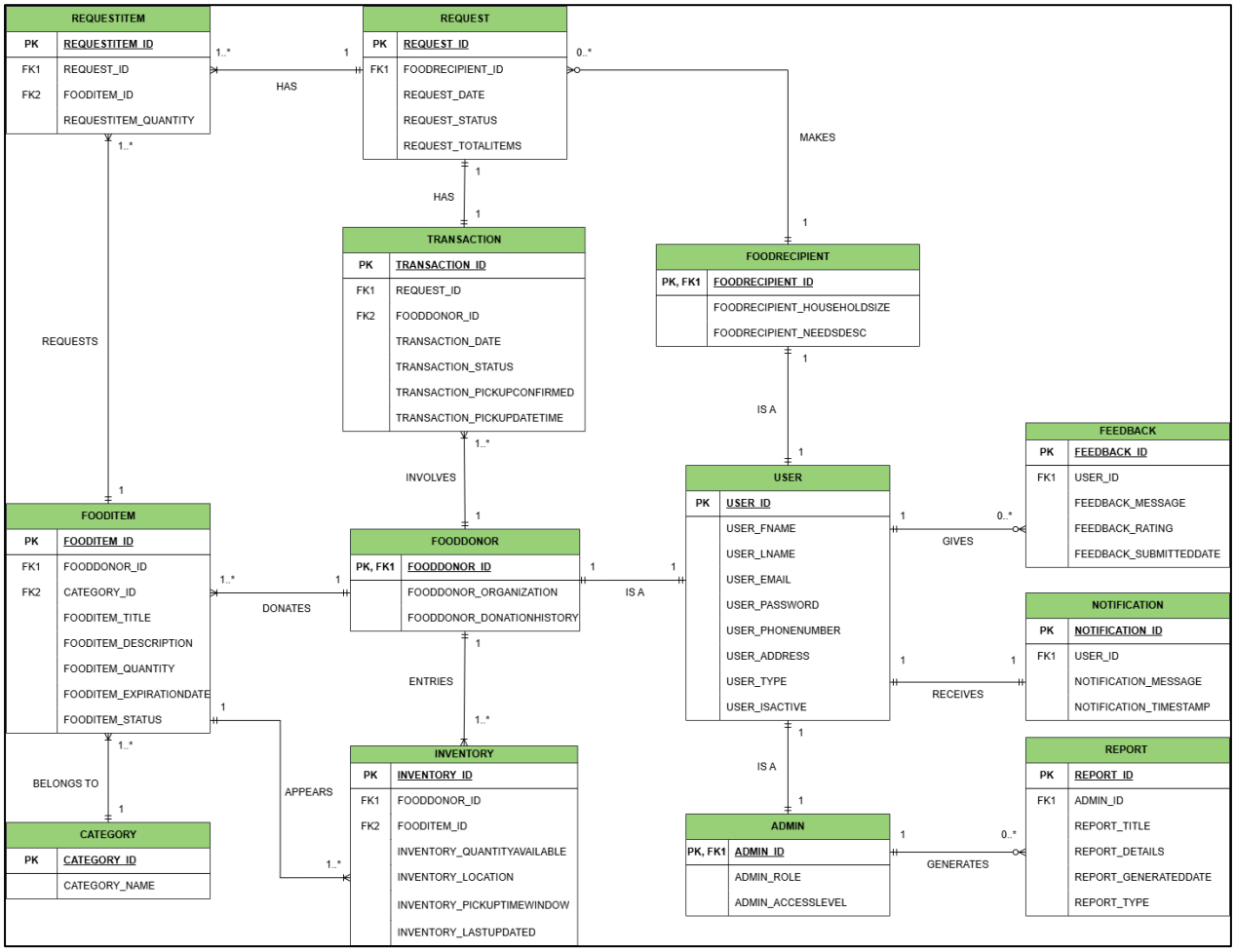

Figure 3: 2.3.1 – Entity-Relationship Diagram of Food Donation & Redistribution System

Table 2: 2.3.1 – System Entities and Attributes Overview

ENTITIES	ATTRIBUTES	DESCRIPTION
USER	**USER_ID (PK)**	Unique identifier for each user
	USER_FNAME	First name of the user
	USER_LNAME	Last name of the user
	USER_EMAIL	Email address of the user (for login and communication)
	USER_PASSWORD	Encrypted password for security purposes
	USER_PHONENUMBER	Contact phone number of the user
	USER_ADDRESS	Home address of the user
	USER_TYPE	Specifies whether the user is a Donor, Recipient, or Admin
	USER_ISACTIVE	Indicates if the user's account is active
FOODDONOR	**FOODDONOR_ID (PK)**	Unique identifier for each food donor
	FOODDONOR_ORGANIZATION	Name of the donor organization (if applicable)
	FOODDONOR_DONATIONHISTORY	Summary or record of previous donations
FOODRECIPIENT	**FOODRECIPIENT_ID (PK)**	Unique identifier for the food recipient
	FOODRECIPIENT_HOUSEHOLDSIZE	Number of people in the recipient's household
	FOODRECIPIENT_NEEDSDESC	Description of the food needs or preferences

ADMIN	ADMIN_ID (PK)	Unique identifier for each administrator
	ADMIN_ROLE	Role or position of the admin
	ADMIN_ACCESSLEVEL	Defines the admin's level of access and privileges in the system
CATEGORY	**CATEGORY_ID (PK)**	Unique identifier for the category of each food item
	CATEGORY_NAME	Name of the category
FOODITEM	**FOODITEM_ID (PK)**	Unique identifier for each food item
	FOODDONOR_ID (FK1)	Foreign key linking to the FoodDonor providing the food item
	CATEGORY_ID (FK2)	Foreign key linking to the Category, providing the food category.
	FOODITEM_TITLE	Name/Title of the food item
	FOODITEM_DESCRIPTION	Description/details of the food item
	FOODITEM_QUANTITY	Quantity available for donation
	FOODITEM_EXPIRATIONDATE	Expiration date of the food item
	FOODITEM_STATUS	Status of the food item (Available, Reserved, Expired)
	REQUEST_ID (PK)	Unique identifier for each food request
	RECIPIENT_ID (FK1)	Foreign key linking to the FoodRecipient (User)

REQUEST	REQUEST_DATE	The date when the request was made
	REQUEST_STATUS	Current status of the request
	REQUEST_TOTALITEMS	Total number of food items requested
REQUESTITEM	**REQUESTITEM_ID (PK)**	Unique identifier for each request item
	REQUEST_ID (FK1)	Foreign key linking to the related Request
	FOODITEM_ID (FK2)	Foreign key linking to the related FoodItem
	REQUESTITEM_QUANTITY	Quantity of the food item requested
INVENTORY	**INVENTORY_ID (PK)**	Unique identifier for each inventory entry
	FOODDONOR_ID (FK1)	Foreign key linking to the FoodDonor
	FOODITEM_ID (FK2)	Foreign key linking to the FoodItem
	INVENTORY_QUANTITYAVAILABLE	Current quantity of the inventory item available
	INVENTORY_LOCATION	Physical location where the food is stored
	INVENTORY_PICKUPTIMEWINDOW	Time window in which food can be picked up
	INVENTORY_LASTUPDATED	Date the inventory was last updated
	TRANSACTION_ID (PK)	Unique identifier for each transaction record
	REQUEST_ID (FK1)	Foreign key linking to the associated Request
	FOODDONOR_ID (FK2)	Foreign key linking to the associated FoodDonor

TRANSACTION	TRANSACTION_DATE	The date when the transaction was initiated
	TRANSACTION_STATUS	Current status of the transaction
	TRANSACTION_PICKUPCONFIRMED	Boolean flag indicating if the pickup is confirmed
	TRANSACTION_PICKUPDATETIME	Scheduled date and time for pickup
FEEDBACK	**FEEDBACK_ID (PK)**	Unique identifier for each feedback record
	USER_ID (FK1)	Foreign key linking to the user who submitted the feedback
	FEEDBACK_MESSAGE	Content of the feedback given by the user
	FEEDBACK_RATING	Rating score given by the user
	FEEDBACK_SUBMITTEDDATE	Date when the feedback was submitted
NOTIFICATION	**NOTIFICATION_ID (PK)**	Unique identifier for each notification entry
	USER_ID (FK1)	Foreign key linking to the intended user
	NOTIFICATION_MESSAGE	Content of the notification sent to the user
	NOTIFICATION_TIMESTAMP	Time when the notification was generated
	REPORT_ID (PK)	Unique identifier for each report

REPORT	ADMIN_ID (FK1)	Foreign key linking to the Admin who created the report
	REPORT_TITLE	Title of the report
	REPORT_DETAILS	Detailed content of the report
	REPORT_GENERATEDDATE	Date the report was created
	REPORT_TYPE	Type/Category of the report

2.4 Object Diagram

This object diagram illustrates a snapshot of the Food Redistribution & Donation System in operation, demonstrating instantiated objects and their relationships. It showcases a user hierarchy with a base User class and specialized roles like FoodDonor, FoodRecipient, and Admin, highlighting the use of inheritance. The diagram details a specific donation of a "Burger," its inventory status and location, a recipient with dietary needs, and a completed transaction for that food item. Furthermore, it includes a food request, associated request item, user feedback on a past interaction, a successful donation notification sent to the donor, and an analytical report on customer feedback trends.

The depicted objects and their connections reveal key aspects of the system's behaviour, including user interactions, food item management, transaction processing, feedback collection, and reporting capabilities. The strong associations between entities like Transaction, Request, RequestItem, Feedback, and Notification with User and FoodItem instances emphasize the interconnectedness of the system's components. The diagram effectively demonstrates how real-time data is managed to facilitate food donation, track inventory, process requests, gather user feedback, and provide administrative oversight and analytical insights, reflecting a domain-driven design approach.

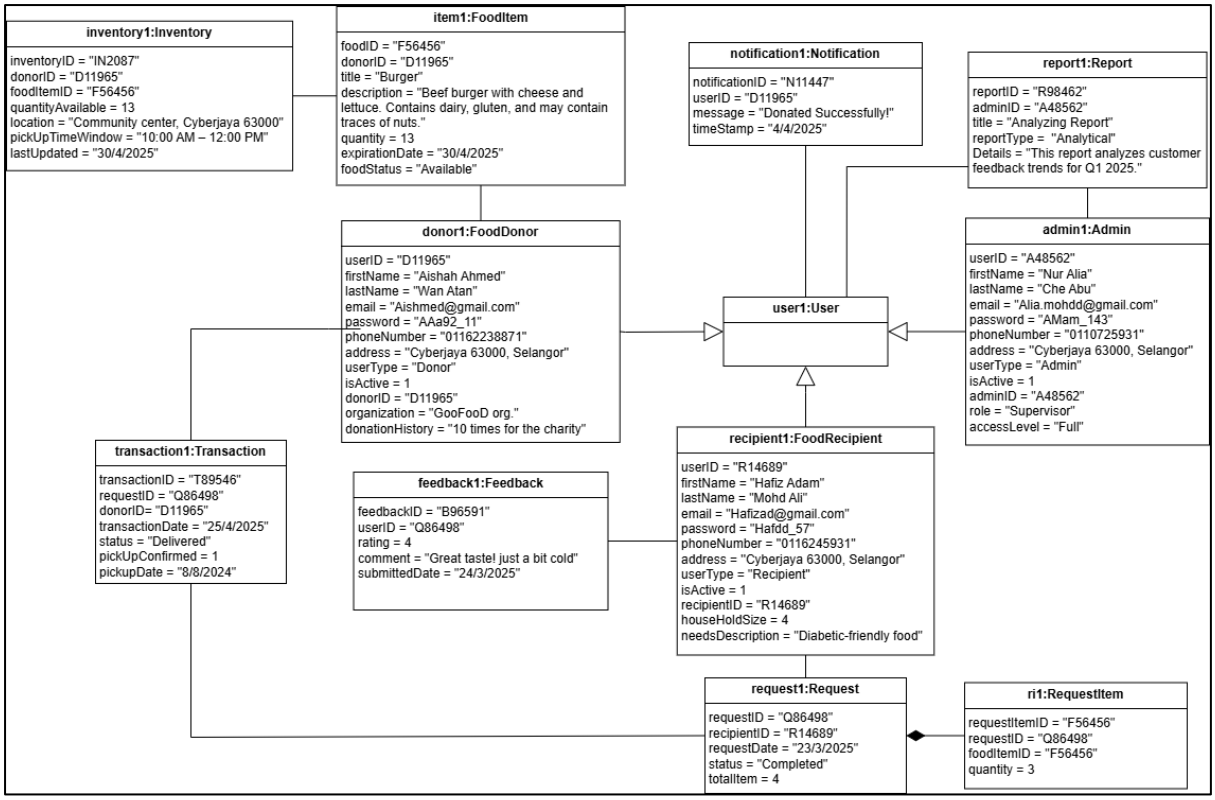

Figure 4: 2.4.1 – Object Diagram for Food Donation and Redistribution System

3.0 Functional and Non-Functional Requirements

The Food Redistribution and Donation System is designed to meet specific functional and non-functional requirements to ensure usability, efficiency, security, and scalability. The following sections detail these requirements clearly and systematically:

Table 3: 3.0.1 – Functional Requirements for Food Donation and Redistribution System

	Functional Requirements
FR1	The system shall allow food donors to register, log in securely, and manage food item listings (create, update, delete).
FR2	The system shall enable food recipients to browse available food listings, submit food requests, and track request statuses.
FR3	The system shall allow administrators to manage user accounts, moderate food listings, oversee transactions, and generate activity reports.
FR4	The system shall send real-time notifications to users regarding food approvals, request updates, and pickup schedules.
FR5	The system shall maintain donation histories and request tracking for users.
FR6	The system shall allow recipients to cancel food requests, and donors to withdraw food listings.
FR7	The system shall enforce secure role-based access control, ensuring each user group has appropriate permissions.
FR8	The system shall provide a platform for submitting feedback from donors and recipients.

Table 4: 3.0.2 – Non-Functional Requirements for Food Donation & Redistribution System

	Non-Functional Requirements
NFR1	**Performance:** The system shall respond within 2 seconds for key operations (e.g., login, browse, submit request).
NFR2	**Scalability:** The system shall support at least 1,000 concurrent users under normal operating conditions.
NFR3	**Security:** All user data must be protected via secure authentication, encryption, and controlled access.
NFR4	**Usability:** The system shall provide a user-friendly, intuitive interface compatible with desktop and mobile browsers.
NFR5	**Availability:** The system shall maintain a minimum of 99% uptime, excluding scheduled maintenance windows.
NFR6	**Maintainability:** The system must adopt a modular, object-oriented design to facilitate easy updates and enhancements.
NFR7	**Data Integrity:** Consistency and accuracy of inventory, transaction, and user data must be ensured at all times.
NFR8	**Compliance:** The system shall follow basic user data protection principles similar to GDPR standards.

3.1 Software Design Concepts

This part explains the fundamental software design principles that form the foundation of the Food Redistribution & Donation System. Principles such as modularity, abstraction, encapsulation, separation of concerns, and reusability are employed to make the system organized, maintainable, and scalable. Each principle is explained with its actual

implementation in the system to enable good component organization, clean architectural practices, and the establishment of a sound, extensible platform.

3.1.1 Modularity

Modularity refers to the organization of a system into separate, independent components that have defined responsibility. The Food Redistribution & Donation System applies the modular organization in that it organizes the platform as function-specific and role-specific modules.

The system is structured around its three main user roles: Food Donor, Food Recipient, and Admin, each accessing only the modules relevant to their responsibilities. For instance:

The **User Management Module** handles registration, login, role-based access control, and profile updates for all user types.

The **Donation Management Module** allows donors to list, edit, and delete surplus food items.

The **Request Management Module** enables recipients to browse available food, submit requests, track status, and manage pickups.

The **Pickup Scheduling Module** coordinates pickup details between donors and recipients.

The Notification Module dispatches real-time alerts triggered by system events such as approvals and pickup confirmations.

The **Feedback Module** collects recipient feedback following successful transactions.

The **Admin Tools Module** provides functionalities for managing users, moderating requests, generating reports, and configuring system rules.

The **Inventory & Transaction Module** records quantities of available food, transaction details, and pickup history.

Each module is loosely coupled and interacts through clearly defined interfaces, enabling independent development, testing, and deployment.

Modularity improves maintainability since changes in one module do not affect others. It improves collaborative development since multiple developers can develop individual modules at the same time. It also provides scalability, where new features—such as volunteer integration or advanced analytics—can be added without disrupting existing processes.

3.1.2 Abstraction

Abstraction is a fundamental concept in object-oriented design that focuses on simplifying complex systems by modelling classes based on the essential properties and behaviours of real-world entities while hiding unnecessary details. This design concept allows developers to

create a clear interface for interacting with objects, enabling them to work with higher-level concepts without needing to understand the complicated workings of the underlying implementation.

In the Food Redistribution and Donation System, abstraction effectively applies across User, FoodItem, and Request class to streamline interactions and encapsulate complex logic.

Abstraction in the User class

The User class abstracts user-related data and behaviours, such as role and phone number. It provides a simplified interface for accessing user details without exposing the underlying complexities of user management. For Instance, methods like `getRole()` and `setPhoneNumber(String phoneNumber)` allow external components to interact with user data while keeping the implementation details hidden.

Abstraction in the FoodItem class

The FoodItem class encapsulates attributes related to food donations, such as title, description, quantity, and status. It provides methos like `isExpired()` and `updateStatus(String newStatus)` that abstract the logic for checking donation validity and updating its state. This allows other components to interact with the food donations without needing to understand the underlying data structure or validation rules.

Abstraction in the Request class

The Request class abstract the process of managing donation requests. It includes methods like `approveRequest()` and `rejectRequest()`, which encapsulate the logic for handling request statuses. This abstraction allows external components to manage donation requests without delving into the details of how requests are processed or stored.

The Food Redistribution and Donation System achieves a clean and modular architecture by adhering to the principle of abstraction. The design approach enhances code maintainability by allowing developers to modify internal implementations without affecting external interactions. It also improves scalability, as new features can be added through extended classes or methods without altering existing interfaces

3.1.3 Encapsulation

Encapsulation is one of the key principles of object-oriented design, in which the data (attributes) and behaviour (methods) of an object are bundled together, and access to its internal state is denied directly. This practice enhances data integrity, promotes maintainability, and makes internal implementation changes do not affect external components. Encapsulation also facilitates abstraction by revealing only the required details through controlled interfaces.

In the Food Redistribution and Donation System, encapsulation is rigorously applied across key classes—User, FoodItem, Request, Transaction, and Notification—to enforce security, validation, and modularity.

Encapsulation in the User Class

Sensitive properties such as password, email, phoneNumber, and role are wrapped in the User class. External direct access is limited by using getter and setter methods (i.e., `getEmail()`, `setRole(String role)`). For example, password modification must use the `changePassword(newPassword: String)` method so that any encryption or validation rules are applied centrally.

This design enforces authentication sequences and follows the best practices on data privacy as well as managing user accounts.

Encapsulation in the FoodItem Class

The FoodItem class encapsulates key attributes such as `quantity`, `expirationDate`, and `foodStatus`. These are accessed and mutated via dedicated methods like `setQuantity(int quantity)` and `updateStatus(String newStatus)`. Behavioral logic, such as determining if a food item has expired, is encapsulated within the method `isExpired()`, thereby shielding external modules from the complexity of date-based logic.

Encapsulation in the Transaction Class

The Transaction class uses encapsulation to manage delivery lifecycle attributes such as `deliveryStatus` and `collectedBy`. These are only modified through validated methods like `assignCollector(String name)` and `updateDeliveryStatus(String status)`. The method `isDelivered()` encapsulates the logic for determining completion based on pickup status.

Encapsulation in the Food Redistribution & Donation System enhances security by preventing direct access to sensitive data such as passwords and user roles. It improves maintainability by localizing logic within classes, allowing changes without impacting external components. Robustness is achieved through controlled state transitions and internal validation, reducing

the likelihood of invalid system behaviour. Additionally, the design supports scalability, as new rules (e.g., expiry checks or role restrictions) can be added within encapsulated methods without altering external interfaces. Collectively, these advantages contribute to a secure, maintainable, and extensible software architecture aligned with object-oriented best practices.

3.1.4 Separation of Concerns

Separation of Concerns (SoC) is one of the significant design principles that organizes the system into tidily defined parts, with each part addressing a particular concern. SoC in the Food Redistribution & Donation System is implemented at architectural layers, user roles, and modular services to facilitate maintainability, scalability, and secure access control.
At the **architectural level**, the system adopts Django's Model-View-Template (MVT) framework:

- **Models** encapsulate data and business logic (e.g., FoodItem handles expiry checks, Request manages status updates).
- **Views** manage user interactions and business workflows such as submitting requests or approving donations.
- **Templates** handle UI rendering without embedding logic, keeping presentation separate from processing.

At the **role level**, each actor—**Donor**, **Recipient**, and **Admin**—interacts with the system through isolated modules. For example:

- Donors manage food listings (`listFoodItem(), editFoodItem()`).
- Recipients browse and request food (`browseFood(), submitRequest()`).
- Admins moderate requests, manage users, and generate reports (`approveRequest(), generateReport()`).

Additionally, **modular services** like the NotificationManager, MatchingService, and Feedback handler are decoupled from user logic and interface layers. These services use Observer and Strategy Patterns, allowing them to evolve independently of the core application.

The implementation of Separation of Concerns in the system offers several key benefits. It enhances maintainability by ensuring changes in one area, such as notification methods, do not impact unrelated modules. Security is improved with role-specific access, minimizing the risk of unauthorized actions. Scalability is supported by allowing new functionalities, like push notifications or additional user roles, without causing disruptions. Parallel development is facilitated as different teams can independently work on various layers or features.

Overall, this approach makes the system modular, flexible, easy to test, and deploy, ensuring long-term resiliency and sustainability.

3.1.5 Reusability

Reusability is a fundamental software design concept dealing with constructing units of code that can be used more than once within different parts of an application or even different applications. It makes development more productive, reduces development time, ensures consistency, and enhances maintainability.
In our Donation System, reusability is put into practice using several practices:

Modular Code Structure: The system is divided into a number of standalone files such as login.py, donate.py, admin.py, and main.py. Every file performs some tasks (e.g., handling login, donation operations, and admin management), thus, they can be reused or updated without affecting the entire system.
Reusable Functions: Common tasks such as validating user inputs, displaying menus, and processing donations are encapsulated into functions. Example:

```
def login():
    # login logic
```

The `login()` function can be called whenever a login operation is needed, without rewriting the logic every time.

Separation of User Roles: By segregating user functionalities (Donor, Recipient, and Admin) into different modules, I can use the login template and donation functionality for all roles without redundancy.
Standard Templates: Common outputs like menu displays, confirmation messages, and error messages are standardized. The standardization enables easy reuse of the output appearance across different operations.
Future Reusability: As the system modules (e.g., login, donation processing) are separate, they would easily be reused in future applications like fundraising applications, membership systems, or any site requiring user authentication and transaction history.

3.2 Software Design Principles

The Food Redistribution & Donation System architecture relies on established software design principles that promote modularity, maintainability, and long-term scalability. Central among these is compliance with the SOLID principles that guide component structuring in a manner where components are single-missioned, extensible, and loosely coupled. The principles are utilized throughout the system's

architecture, from role-based functionality, request handling workflows, notification management, and service integration. Following these principles, the system achieves good cohesion within separate modules and minimal dependency among them, making them easily testable, independently updateable, and seamlessly feature-expandable. The principled pattern design promotes building a clean, stable, and extensible platform that can change to meet emerging user needs and technological advancements.

3.2.1 Single Responsibility Principle (SRP)

The Single Responsibility Principle (SRP) states that a class or module should have one, and only one, reason to change, i.e., be responsible for a single, coherent responsibility. SRP is followed rigorously throughout the Food Redistribution & Donation System so that each component addresses one specific concern, thereby improving modularity, testability, and maintainability.

For instance:

- The **User class** is solely responsible for user-related data and actions such as login, registration, and profile updates. It does not handle food listings or notifications.
- The **FoodItem class** manages attributes and behaviours related to food inventory (e.g., `setQuantity(), isExpired()),` without being involved in request processing or delivery tracking.
- The **Request class** handles the lifecycle of a food request, including status updates, cancellations, and item associations.
- The **Notification** is dedicated to sending user notifications and is isolated from core business logic.
- The **MatchingStrategy** encapsulates the logic for matching food items to recipients using pluggable strategies (e.g., by expiry or location), separated from data storage or user management.

Each service or class changes only when its own business rules change. For example, updating the notification delivery mechanism does not affect food listing or request workflows, thanks to strict adherence to SRP.

Adhering to the Single Responsibility Principle (SRP) has several advantages. It improves maintainability by keeping updates or bug fixes within a single module. Testing is easier, as units with a single responsibility are easier to test and isolate. Scalability is improved, as it becomes simpler to add new features without changing unrelated classes. In addition, SRP reduces the risk of side effects because single-point modifications minimize the chance of

unforeseen effects elsewhere in the system. Through strict adherence to SRP, the system architecture is clean, modular, and solid—vital attributes for enduring software development and effective team collaboration.

3.2.2 Open/Closed Principle (OCP)

The Open/Closed Principle (OCP) states that software entities (such as classes, modules, and functions) should be open for extension but closed for modification. This means that the behaviour of a module can be extended without modifying its source code, allowing for new functionality to be added while preserving existing code. OCP promotes the use of interfaces or abstract classes, enabling developers to introduce new implementations without altering the existing code.

In the context of the Food Redistribution & Donation System, the OCP is adhered to be designing components that can be extended through inheritance or composition, rather than requiring changes to the existing code.

For instance:

- The **AdminActivity** model encapsulates the actions performed by administrators, such as approving or rejecting donations. If new actions need to be added in the future, such as adding additional types of admin activities, this can be achieved by extending the model or creating new subclasses without modifying the existing “AdminActivity” model.
- The **DonationRequest** model handles the lifecycle of donation requests. If new request types or statuses are introduced, they can be implemented as new subclasses or through additional methods, allowing the existing request handling logic to remain unchanged.
- The **Profile** model manages user profiles, including roles and associated data. If new user roles are required, they can be added through subclassing or by extending the role management logic without altering the existing profile structure.

3.2.3 Liskov Substitution Principle (LSP)

The Liskov Substitution Principle (LSP) is reflected in the way different user roles are handled between admin users and donors. Each of these user groups can interact with the system by logging in, accessing features, and performing actions specific to their roles. Although there are other operations available for admins and donors, the system treats them under a common expectation: any user, whether admin or donor, should be able to log in and perform their

allowed actions without causing the system to behave incorrectly. If I were to further develop the system by introducing subclasses for different user types (e.g., AdminUser and DonorUser classes), each subclass would be able to substitute the general user type without changing the functionality of the system. This principle of design ensures that the system is robust and that new user types can be added in the future without disrupting existing functionality, being true to the LSP spirit.

3.2.4 Dependency Inversion Principle (DIP)

The Dependency Inversion Principle (DIP) is implemented subtly such that high-level modules do not depend on low-level modules, but both depend on abstractions. For instance, the main.py module is a high-level module that controls the application flow, while lower-level modules like login.py, donate.py, and admin.py control functions. Instead of dumping the complex login, donation, or admin behaviour into the main control flow, the system calls login(), donate(), and admin() functions from separate, independent modules. This means the main application is not reliant on the exact implementation details; it just needs to know that these modules provide the necessary functionality regarding known interfaces (functions). As a result, if the internal workings of, for instance, the login() function need to be changed (e.g., from file-based to database-based authentication), this can be done without changing the main program's logic. This separation improves flexibility, maintainability, and scalability because modules can develop separately as long as their interfaces remain constant.

3.2.5 Interface Segregation Principle (ISP)

The Interface Segregation Principle (ISP) allows one to design role-based interfaces where the users handle operations relevant to their responsibilities. In Food Redistribution & Donation System, ISP is implemented for the three key roles of the platform—Food Donor, Food Recipient, and Admin—to ensure clarity, efficiency, and security.

- **Food Donors** access functionalities such as registering, managing their profiles, listing food items, editing listings, viewing request history, scheduling pickups, and submitting feedback.
- **Food Recipients** interact with interfaces to register, browse available donations, submit and cancel requests, track request statuses, manage pickup schedules, confirm pickups, and submit feedback.
- **Administrators** utilize a dedicated interface to manage user accounts, approve or reject donation requests, monitor donation and pickup activity, configure system rules, and generate reports.

Each user role is assigned only the methods it needs—such as approveRequest() for Admins or submitRequest() for Recipients—preventing exposure to irrelevant system functions.

Using ISP results in cleaner, role-focused interfaces, minimizing the risk of user error and reducing cognitive complexity. It reinforces secure access control by preventing users from invoking unauthorized operations. Furthermore, it simplifies testing and future extension of the system by isolating changes to specific role interfaces.

4.0 Proposed Design Patterns

4.1 Factory Design Pattern

4.1.1 Function or Software Component Affected

- UserManager (custom Django manager)
- User model (abstract base class for Donor, Recipient, Admin)
- Subclasses of User determined by user_type (via User.UserType)

This pattern centralizes user creation logic in UserManager, abstracting the instantiation of various user types like Donor, Recipient, and Admin based on role data.

4.1.2 Description of workflow or data flow

1. Client (e.g., Registration View) triggers user creation by invoking UserManager.create_user().
2. The function sanitizes input and infers user type from the user_type field.
3. Creates a User object and fills it with normalized data.
4. Hashes password securely using set_password().
5. Saves user to database using save(using=self._db).
6. Returns created user object back to client.
7. Promotes encapsulation, makes user creation logic reusable, and uniform across modules.

4.1.3 Sample Class Diagram

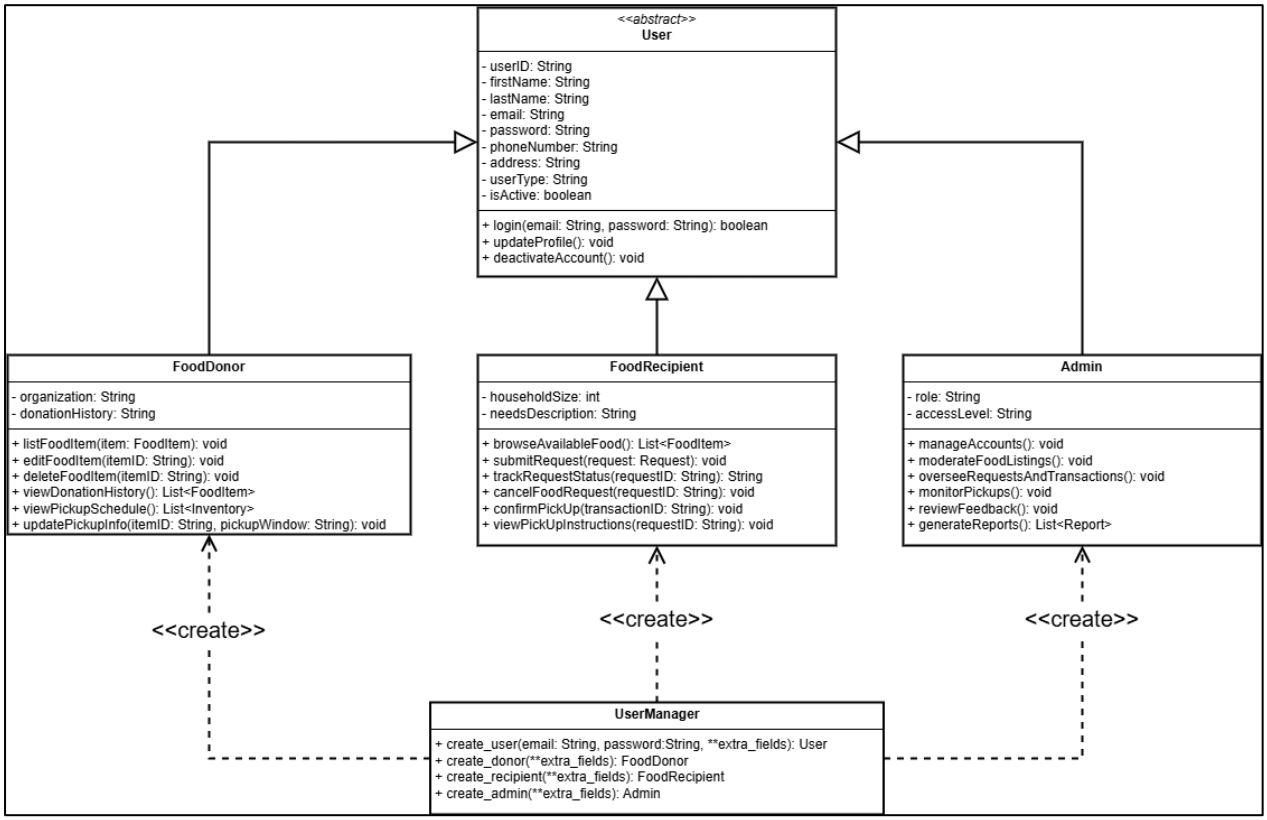

Figure 5: 4.1.3.1 – Factory Design Pattern Class Diagram

4.1.4 Sample Activity Diagram

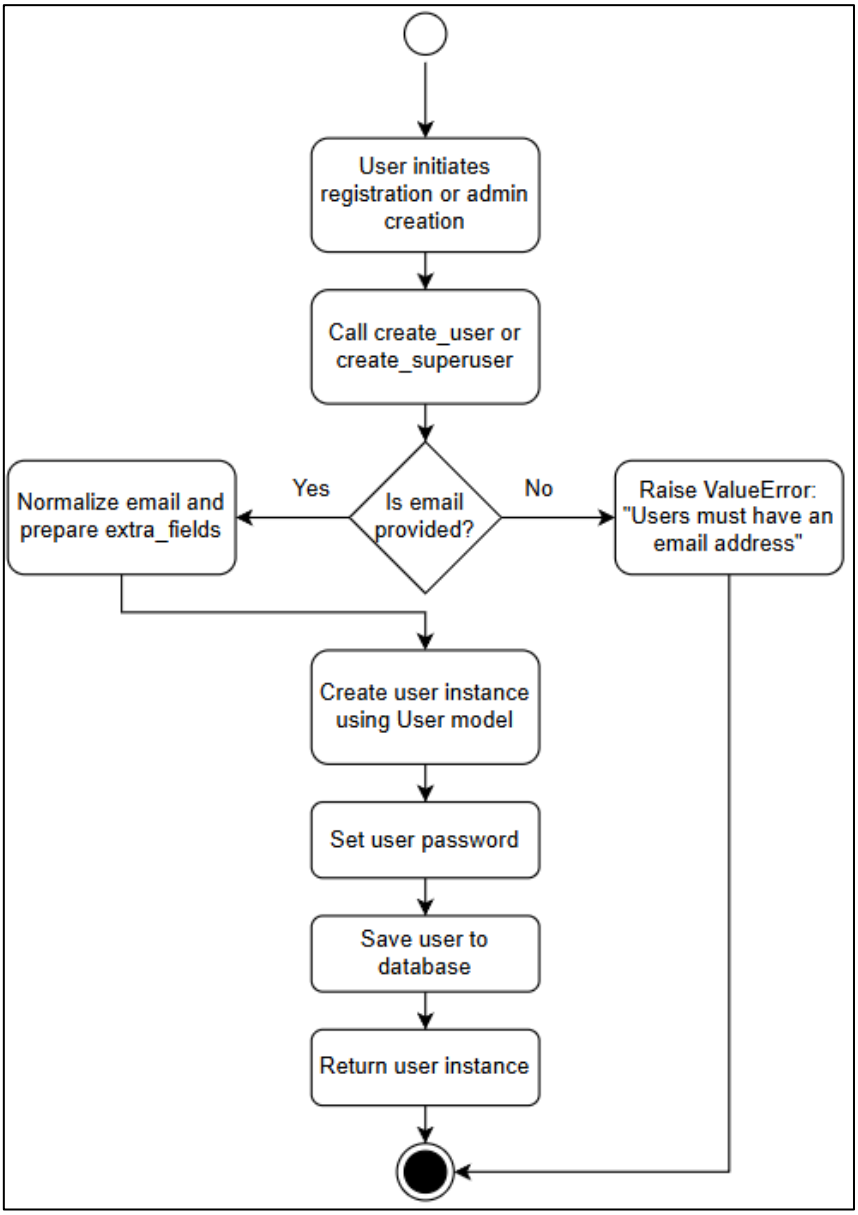

Figure 6: 4.1.4.1 – Factory Design Pattern Activity Diagram

4.1.5 Sample Sequence Diagram

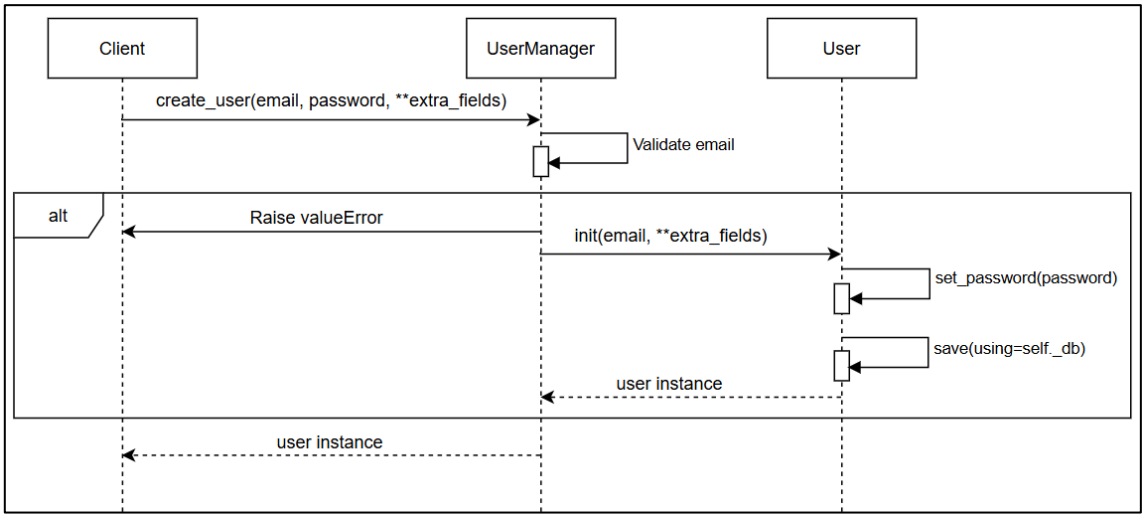

Figure 7: 4.1.5.1 – Factory Design Pattern Sequence Diagram

4.1.6 Sample Potential Code

```
## ===== Factory Pattern Implementation =====
class UserManager(BaseUserManager):
    """Factory pattern for user creation"""
    def create_user(self, email, password=None, **extra_fields):
        if not email:
            raise ValueError('Users must have an email address')

        user = self.model(
            email=self.normalize_email(email),
            **extra_fields
        )
        user.set_password(password)
        user.save(using=self._db)
        return user

    def create_superuser(self, email, password=None, **extra_fields):
        extra_fields.setdefault('is_staff', True)
        extra_fields.setdefault('is_superuser', True)
        extra_fields.setdefault('user_type', User.UserType.ADMIN)
        return self.create_user(email, password, **extra_fields)

class User(AbstractBaseUser):
    """Base user model using Factory pattern for different user types"""
    class UserType(models.TextChoices):
        DONOR = 'DONOR', 'Food Donor'
        RECIPIENT = 'RECIPIENT', 'Food Recipient'
        ADMIN = 'ADMIN', 'System Admin'

    user_id = models.AutoField(primary_key=True)
    email = models.EmailField(unique=True)
    first_name = models.CharField(max_length=100)
```

```
    last_name = models.CharField(max_length=100)
    phone_number = models.CharField(max_length=20)
    address = models.TextField()
    user_type = models.CharField(max_length=10, choices=UserType.choices)
    is_active = models.BooleanField(default=True)
    date_joined = models.DateTimeField(auto_now_add=True)
    last_login = models.DateTimeField(auto_now=True)
    is_staff = models.BooleanField(default=False)
    is_superuser = models.BooleanField(default=False)

    objects = UserManager()

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['first_name', 'last_name', 'phone_number', 'address']

    class Meta:
        db_table = 'users'

    def __str__(self):
        return f"{self.email} ({self.user_type})"

    def has_perm(self, perm, obj=None):
        return self.is_superuser

    def has_module_perms(self, app_label):
        return self.is_superuser

    @property
    def full_name(self):
        return f"{self.first_name} {self.last_name}"
```

4.1.7 Benefits

- You avoid tightly linking the login/register system to specific user types (e.g., recipient, donor, admin), making it easier to manage and extend.
- You follow the Single Responsibility Principle by moving the account creation logic into a dedicated place, keeping your authentication flow clean.
- New user roles or login methods (like social login) can be added without changing existing login/register logic—thanks to the Open/Closed Principle.

4.1.8 Limitations

- It adds extra complexity, especially early on, since you'll need separate classes or methods for each user type. This works best if you're already handling multiple user types or plan to scale.

4.2 Strategy Design Pattern

4.2.1 Function or Software Component Affected

- MatchingStrategy (abstract base class)
- LocationStrategy and ExpiryStrategy (concrete implementations)
- MatchingService (context class using strategies)
- FoodRecipient (matched entity)
- Donation (object to match)

4.2.2 Description of workflow or data flow

The app matches donations to recipients based on multiple criteria: location, food expiry, food requirement, and emergency level. Each of these criteria is implemented as a separate strategy component that contributes to an overall score or matching decision. The Matching Service executes all strategies sequentially and aggregates the results.

Workflow:

- Donor submits a food donation.
- The Matching Service runs all matching strategies.
- Each strategy gives a score or filter result.
- Combined results determine the best recipient(s).
- Assignment is made automatically.

4.2.3 Sample Class Diagram

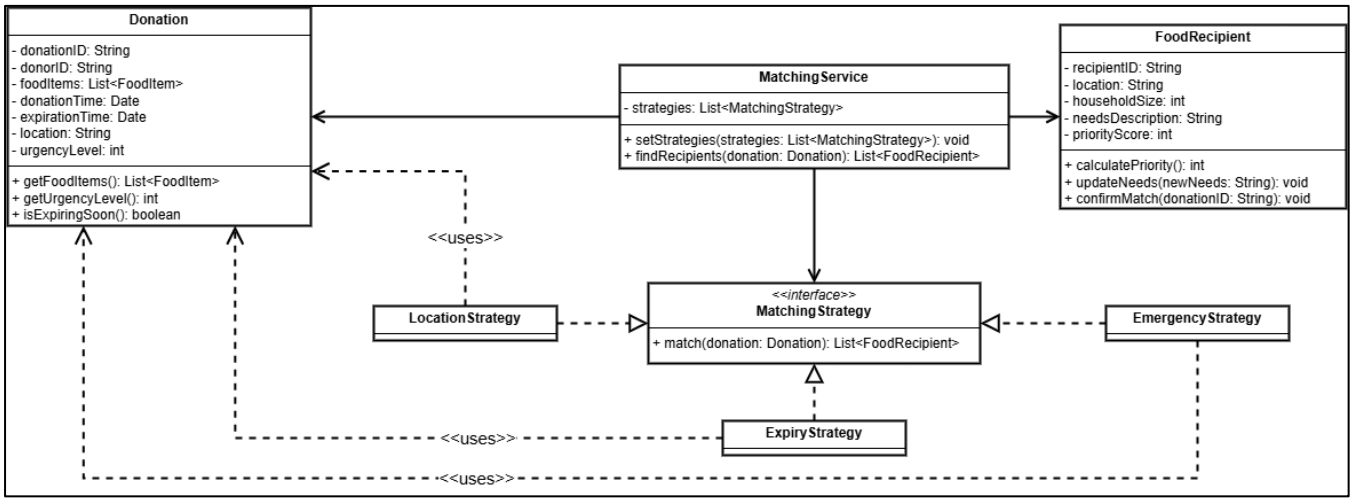

Figure 8: 4.2.3.1 – Strategy Design Pattern Class Diagram

4.2.4 Sample Activity Diagram

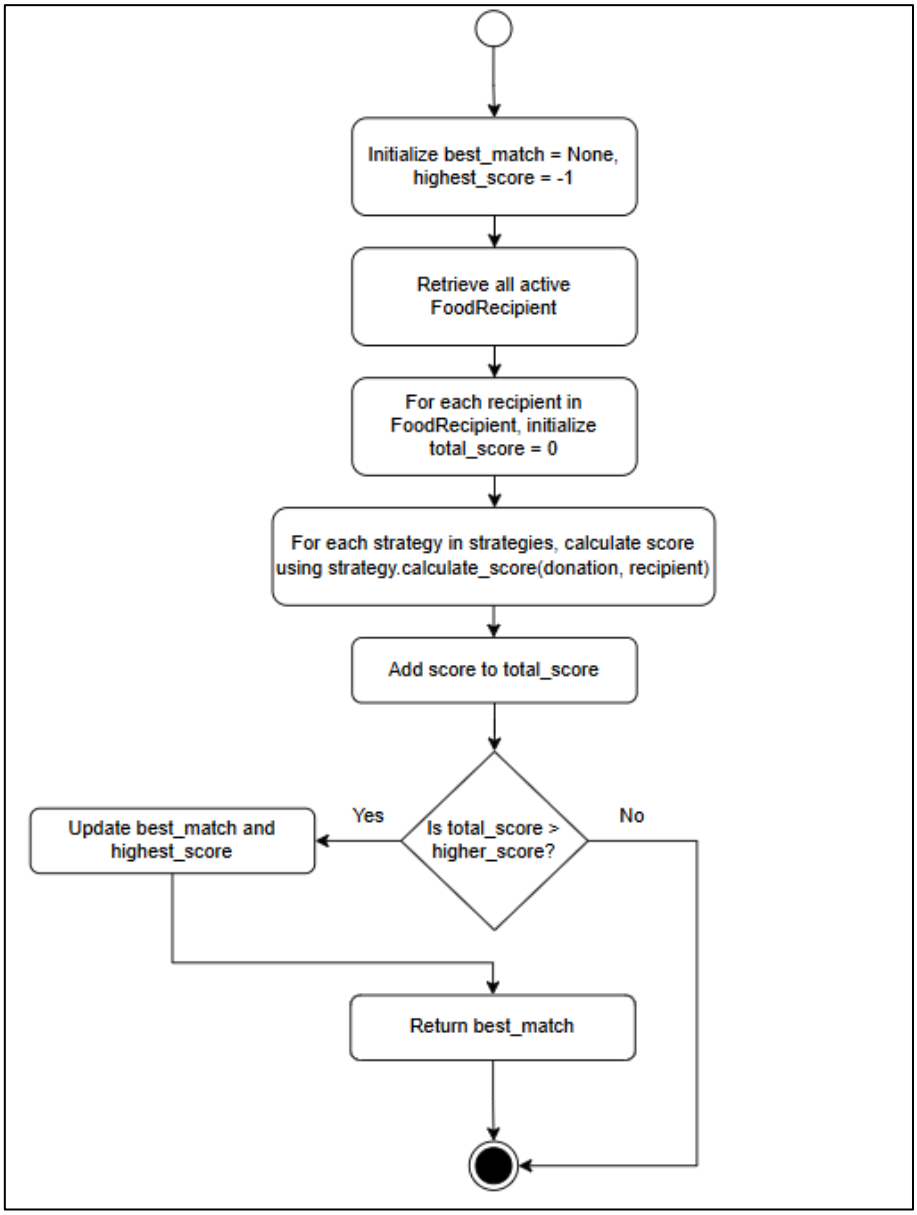

Figure 9: 4.2.4.1 – Strategy Design Pattern Activity Diagram

4.2.5 Sample Sequence Diagram

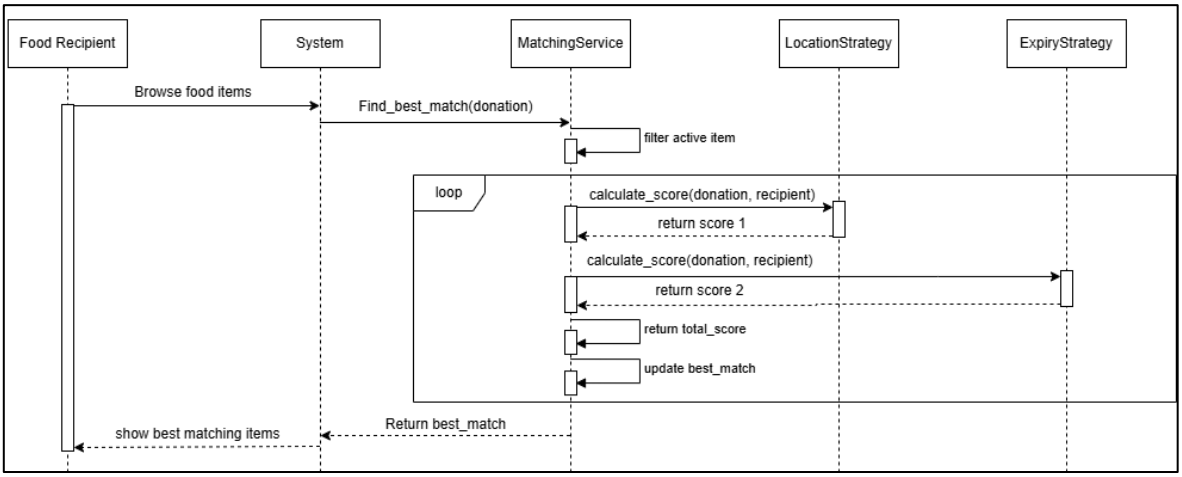

Figure 10: 4.2.5.1 – Strategy Design Pattern Sequence Diagram

4.2.6 Sample Potential Code

```
## ===== Strategy Pattern Implementation =====
class MatchingStrategy(ABC):
    """Strategy interface for matching algorithms"""
    @abstractmethod
    def calculate_score(self, donation, recipient) -> float:
        pass

class LocationStrategy(MatchingStrategy):
    """Strategy for location-based matching"""
    def calculate_score(self, donation, recipient) -> float:
        # Simplified distance calculation - in practice, use geopy or similar
        distance = abs(donation.donor.address - recipient.address)
        return 1 - (distance / 100)  # Assuming max distance is 100km

class ExpiryStrategy(MatchingStrategy):
    """Strategy for urgency-based matching (closer to expiry = higher priority)"""
    def calculate_score(self, donation, recipient) -> float:
        days_remaining = (donation.expiration_date - datetime.now().date()).days
        return 1 - (days_remaining / 30)  # Normalize to 30 days max

class MatchingService:
    """Context class that uses the Strategy pattern"""
    def __init__(self, strategies: List[MatchingStrategy]):
        self.strategies = strategies

    def find_best_match(self, donation) -> 'FoodRecipient':
        from django.db.models import F
        recipients = FoodRecipient.objects.filter(is_active=True)

        best_match = None
        highest_score = -1
```

```
    for recipient in recipients:
        total_score = 0
        for strategy in self.strategies:
            total_score += strategy.calculate_score(donation, recipient)

        if total_score > highest_score:
            highest_score = total_score
            best_match = recipient

    return best_match
```

4.2.7 Benefits

- Each donation method can be implemented separately, keeping the main app logic clean and focused.
- Instead of using inheritance for different donation methods, you can use composition to inject the needed logic.
- The system can easily support new donation options in the future without changing existing code, following the Open/Closed Principle.

4.2.8 Limitations

- If your app only uses one or two donation methods that rarely change, adding extra classes and interfaces may be unnecessary complexity.
- Many modern languages support passing functions directly, so using anonymous functions might be simpler than creating full strategy classes, depending on your needs.

4.3 Observer Design Pattern

4.3.1 Function or Software Component Affected

- Notification System: When a donation is submitted or updated, the system immediately sends out notifications to the donor.
- Donation Workflow: The donation process triggers observer updates to track changes in donation status or log events in real-time.
- User Interface (Dashboard, Logs): The dashboard and logs are **dynamic observers** that refresh or update automatically based on new donations or changes.

4.3.2 Description of workflow or data flow

The Observer Pattern allows different parts of the system to react automatically when certain events take place — without tightly coupling components together.

In this project, when a donation event occurs (or when a user interacts), multiple components like the Notification System, Donation Workflow, and User Interface (Dashboard, Logs) need to be updated at the same time.

- When a donation is made, a DonationManager triggers an event.
- The NotificationManager acts as the Subject in the Observer pattern.
- EmailNotifier, SMSNotifier, and DashboardUpdater act as Observers, each implementing a common interface (e.g., NotificationObserver).
- When a donation occurs, all subscribed observers are notified automatically and can take appropriate actions (send email/SMS, update UI, log events).

4.3.3 Sample Class Diagram

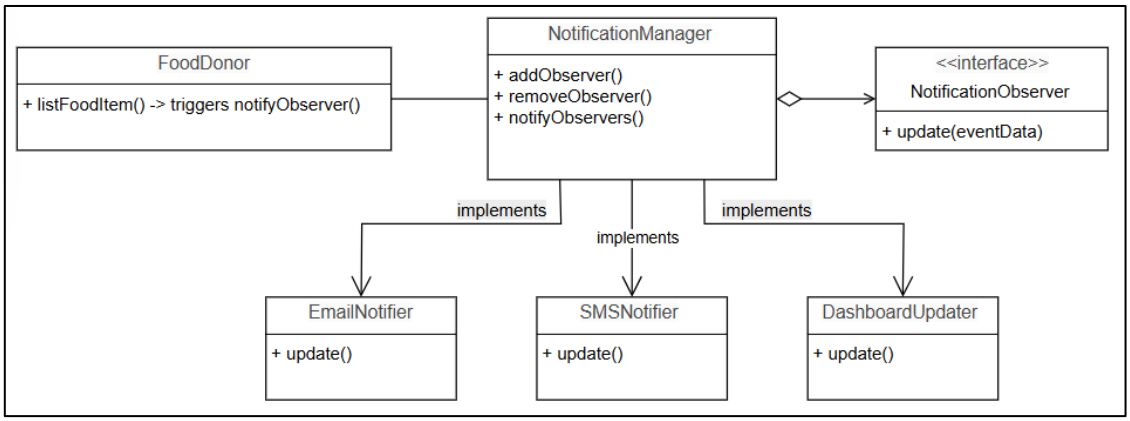

Figure 11: 4.3.3.1 – Observer Design Pattern Class Diagram

4.3.4 Sample Activity Diagram

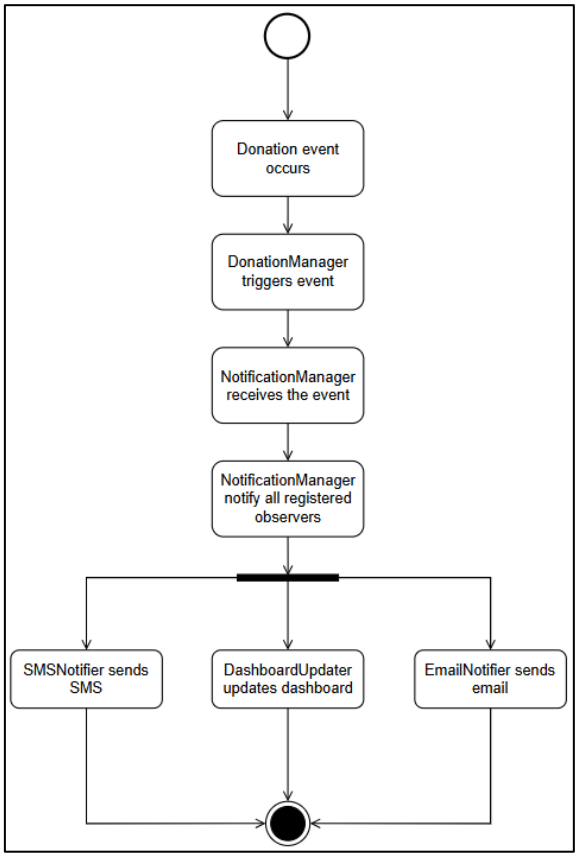

Figure 12: 4.3.4.1 – Observer Design Pattern Activity Diagram

4.3.5 Sample Sequence Diagram

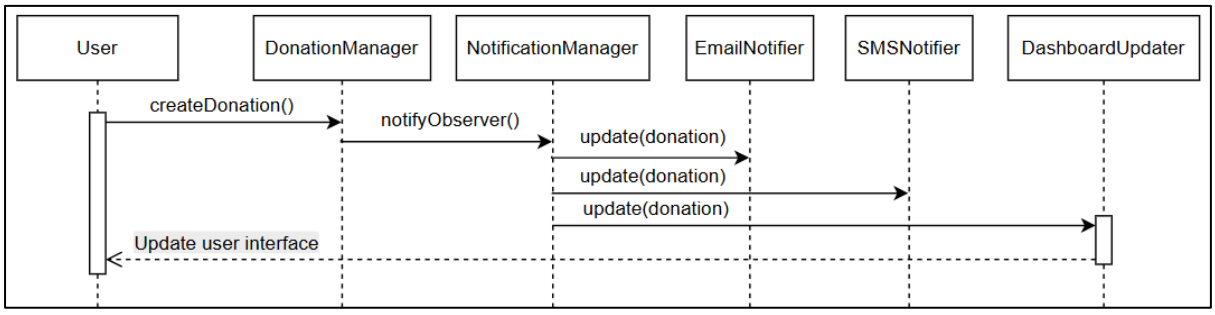

Figure 13: 4.3.5.1 – Observer Design Pattern Sequence Diagram

4.3.6 Sample Potential Code

```
## ===== Observer Pattern Implementation =====
class NotificationSubject:
    """Subject for observer pattern"""
    def __init__(self):
        self._observers = []

    def attach(self, observer):
        self._observers.append(observer)

    def detach(self, observer):
```

```
        self._observers.remove(observer)

    def notify(self, message, user, **kwargs):
        for observer in self._observers:
            observer.update(message, user, **kwargs)

class NotificationObserver(ABC):
    """Observer interface"""
    @abstractmethod
    def update(self, message, user, **kwargs):
        pass

class EmailNotificationObserver(NotificationObserver):
    """Concrete observer for email notifications"""
    def update(self, message, user, **kwargs):
        # In a real implementation, send email here
        Notification.objects.create(
            user=user,
            message=f"Email: {message}",
            timestamp=datetime.now()
        )

class SMSNotificationObserver(NotificationObserver):
    """Concrete observer for SMS notifications"""
    def update(self, message, user, **kwargs):
        # In a real implementation, send SMS here
        Notification.objects.create(
            user=user,
            message=f"SMS: {message}",
            timestamp=datetime.now()
        )

class DashboardUpdater(NotificationObserver):
    """Concrete observer for real-time dashboard updates"""
    def update(self, message, user, **kwargs):
        # Update admin dashboard
        self.update_admin_dashboard(message, user)

        # Update user-specific dashboard if applicable
        if hasattr(user, 'dashboard_preferences'):
            self.update_user_dashboard(user, message)

        # Log the dashboard update
        Notification.objects.create(
            user=user,
            message=f"Dashboard Update: {message}",
            timestamp=datetime.now()
        )

    def update_admin_dashboard(self, message, user):
        """Update the admin dashboard with new activity"""
```

```
        from channels.layers import get_channel_layer
        from asgiref.sync import async_to_sync

        try:
            channel_layer = get_channel_layer()
            event = {
                'type': 'dashboard.message',
                'message': message,
                'user_id': str(user.user_id),
                'timestamp': str(datetime.now()),
                'event_type': kwargs.get('event_type', 'notification')
            }

            async_to_sync(channel_layer.group_send)(
                'dashboard_updates',
                event
            )
        except Exception as e:
            print(f"Failed to update admin dashboard: {str(e)}")

    def update_user_dashboard(self, user, message):
        """Update a specific user's dashboard"""
        # In a real implementation, this would push to the user's personal dashboard

        pass

# Global notification subject
notification_system = NotificationSubject()
notification_system.attach(EmailNotificationObserver())
notification_system.attach(SMSNotificationObserver())
notification_system.attach(DashboardUpdater())
```

4.3.7 Benefits

- Following the Open/Closed Principle, user can add new types of notification handlers (email/SMS) without changing the core donation logic.
- Each part of the system works separately. If user change one part, they usually do not need to change the others.
- When donation is made, all the necessary parts of the system will be updated right away without needing extra work.
- Since each part focuses on doing just one job, it is easier to fix problems or make improvements.
- User can link to different parts of the application, such as dashboards, orders, or logs, so they can respond automatically to donation events while the app is running.

4.3.8 Limitations

- When multiple components are listening for updates, the order they get notified isn't guaranteed, which might lead to unexpected behaviors.
- If there are a lot of updates happening at once, it can slow down the system.

4.4 Singleton Design Pattern

4.4.1 Function or Software Component Affected

- UserAuthManager: ensuring only one UserAuthManager instance manages all authentication operations across the app.

4.4.2 Description of workflow or data flow

The application asks login or register. Instead of generating a fresh authentication manager every time, it loads the one instance of UserAuthManager.
UserAuthManager handles:

- Authentication of credentials
- Storing login sessions
- Signing up for a new user

No matter how many times login/register are called, it leverages one instance only.

4.4.3 Sample Class Diagram

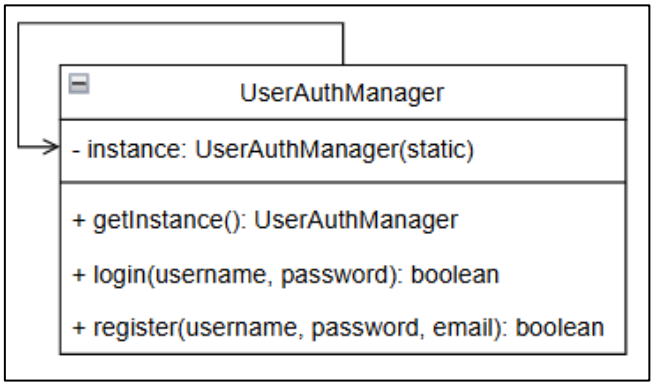

Figure 14: 4.4.3.1 – Singleton Design Pattern Class Diagram

4.4.4 Sample Activity Diagram

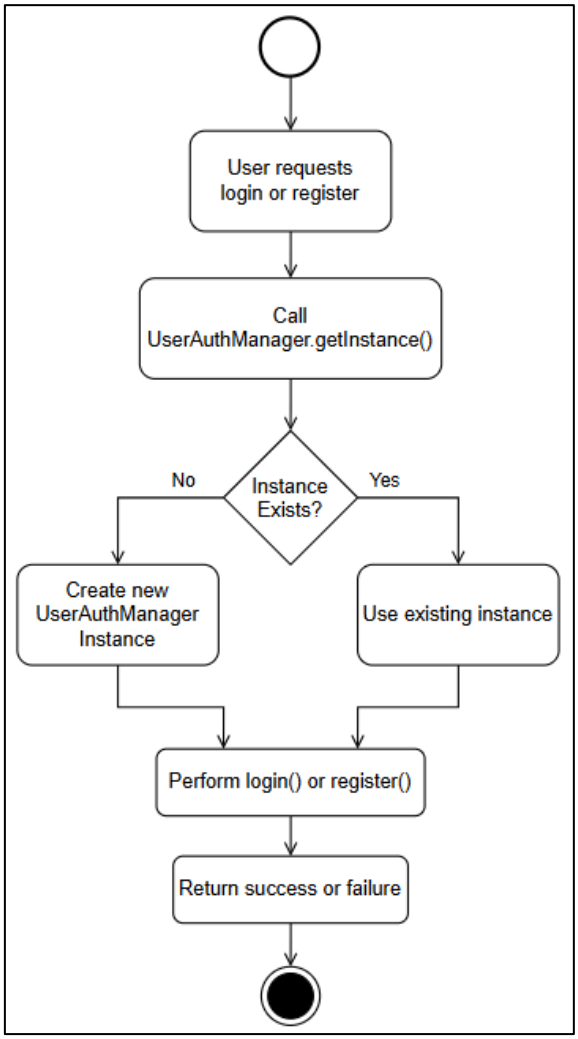

Figure 15: 4.4.4.1 – Singleton Design Pattern Activity Diagram

4.4.5 Sample Sequence Diagram

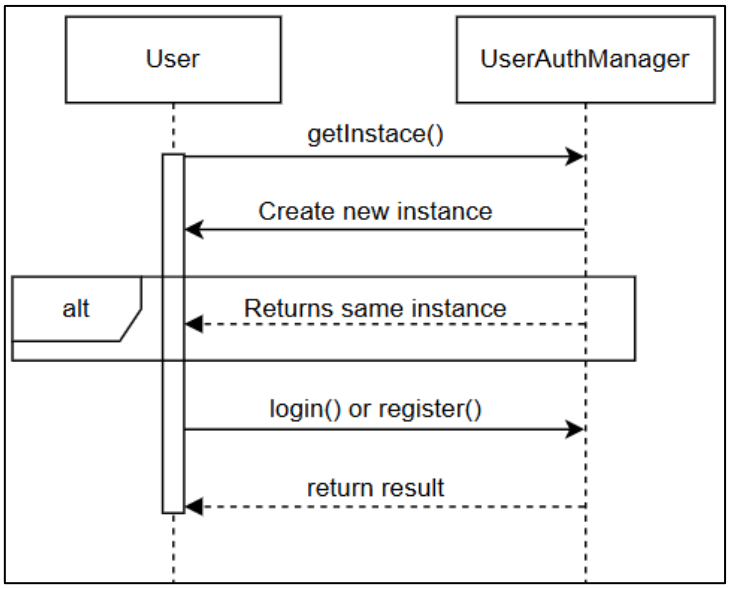

Figure 16: 4.4.5.1 – Singleton Design Pattern Sequence Diagram

4.4.6 Sample Potential Code

```
## ===== Singleton Pattern Implementation =====
class AuthManager:
    """Singleton for authentication management"""
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            cls._instance.active_sessions = {}
        return cls._instance

    def create_session(self, user):
        self.active_sessions[user.user_id] = {
            'user': user,
            'login_time': datetime.now(),
            'last_activity': datetime.now()
        }

    def end_session(self, user_id):
        if user_id in self.active_sessions:
            del self.active_sessions[user_id]

    def get_active_users(self):
        return len(self.active_sessions)
```

4.4.7 Benefits

- You can ensure there's only one instance of a core component.
- That single instance can be accessed globally throughout the app, making it easy to coordinate shared operations.
- The singleton instance is only created when it's needed, saving resources on startup.

4.4.8 Limitations

- It breaks the Single Responsibility Principle since it handles both object creation and global access.
- Singleton usage might hide poor app structure, especially if different parts of the app start relying too much on that one instance.
- In a multithreaded environment, extra care is needed to avoid accidentally creating multiple instances at the same time.

4.5 Decorator Design Pattern

4.5.1 Function or Software Component Affected

The role of the Decorator Pattern is more in the Notification System of the Food Redistribution and Donation System. It ensures that notifications are created and sent to users (Donors, Recipients, Admins) through several different channels-in-app alerts, email, or SMS-without affecting the core notification logic.

4.5.2 Description of workflow or data flow

Basic notification capability (in-app notifications, for example) is dynamically extended with decorators that add more behaviours (sending emails, SMS). The first step after a triggering event (for example, food request approval, or pickup confirmation) is that the system:

1. Creates a basic notification object.
2. Wraps it with one or more decorators (EmailDecorator, SMSDecorator).
3. Calls the send() method, which activates all the notifications behaviours added.

4.5.3 Sample Class Diagram

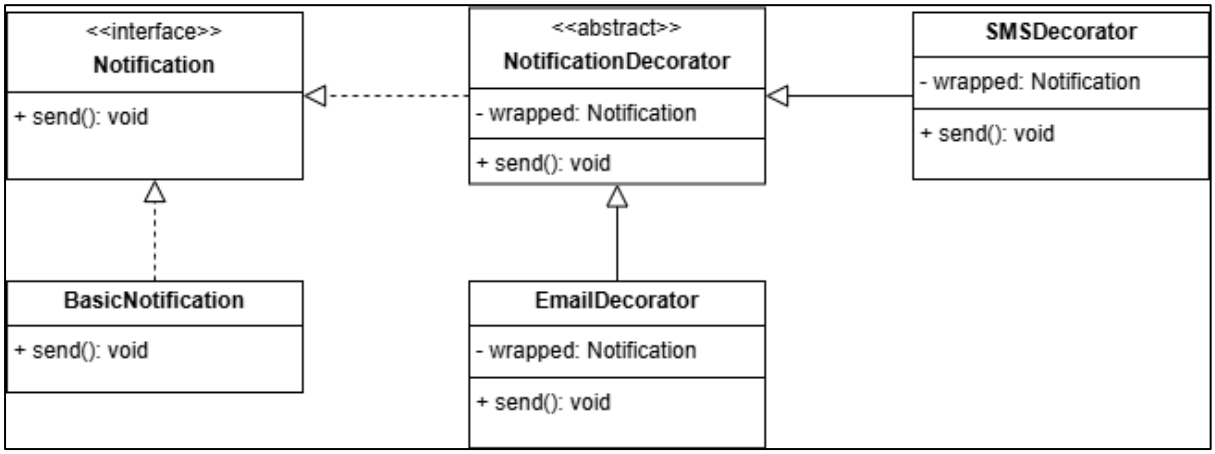

Figure 17: 4.5.3.1 – Decorator Design Pattern Class Diagram

4.5.4 Sample Activity Diagram

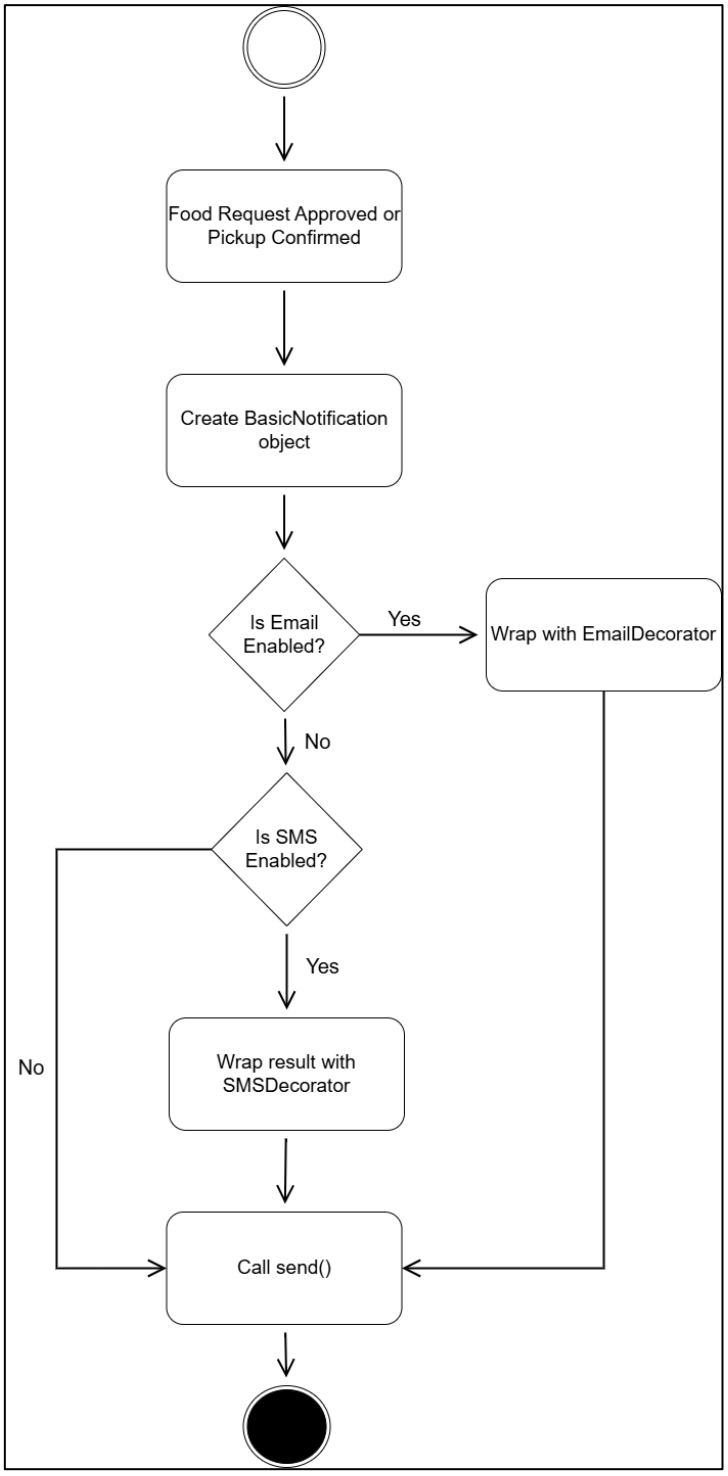

Figure 18: 4.5.4.1 – Decorator Design Pattern Activity Diagram

4.5.5 Sample Sequence Diagram

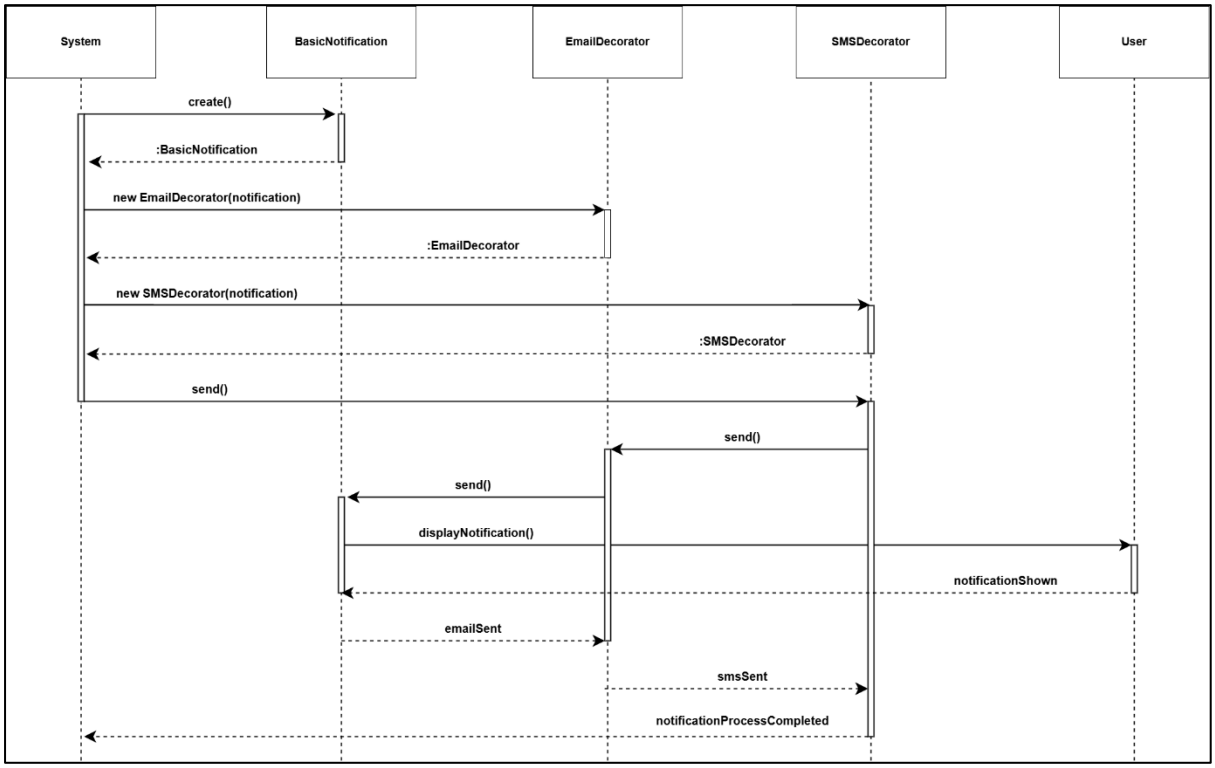

Figure 19: 4.5.5.1 – Decorator Design Pattern Sequence Diagram

4.5.6 Sample Potential Code

```python
# Notification base class (acts as interface)
class Notification:
    def send(self):
        pass  # To be overridden by subclasses

# Basic Notification (In-app alert)
class BasicNotification(Notification):
    def send(self):
        print("Showing in-app notification")

# Abstract Decorator class
class NotificationDecorator(Notification):
    def __init__(self, wrapped):
        self._wrapped = wrapped

    def send(self):
        self._wrapped.send()

# Email Decorator (adds email capability)
class EmailDecorator(NotificationDecorator):
    def send(self):
        super().send()
        print("Email sent to user")

# SMS Decorator (adds SMS capability)
class SMSDecorator(NotificationDecorator):
    def send(self):
        super().send()
        print("SMS sent to user")
```

```python
# Simulated client logic
def notify_user(email_enabled=True, sms_enabled=False):
    # Step 1: Create base notification
    notification = BasicNotification()

    # Step 2: Dynamically add decorators based on settings
    if email_enabled:
        notification = EmailDecorator(notification)
    if sms_enabled:
        notification = SMSDecorator(notification)

    # Step 3: Send notification
    notification.send()

# --- Test Outputs ---

print("Test 1: Email only")
notify_user(email_enabled=True, sms_enabled=False)

print("\nTest 2: SMS only")
notify_user(email_enabled=False, sms_enabled=True)

print("\nTest 3: Email + SMS")
notify_user(email_enabled=True, sms_enabled=True)

print("\nTest 4: In-app only")
notify_user(email_enabled=False, sms_enabled=False)

```

4.5.7 Benefits

- **Flexibility**
  Additional notification types (e.g., Push Notifications) can be added without changing existing code.
- **Single Responsibility Principle (SRP)**
  Each decorator focuses on adding a single type of notification behaviour.
- **Open/Closed Principle (OCP)**
  Classes are open for extension but closed for modification.
- **Scalability**
  The notification system can easily grow to support multiple communication channels.

4.5.8 Limitations

- **Increased Complexity**
  Excessive layering of decorators can make the notification flow difficult to trace and debug.
- **Performance Overhead**

Each decorator adds a level of method call wrapping, slightly increasing processing time.

- **Configuration Management**
  Proper management is needed to ensure correct ordering and combination of notification types.

5.0 Conclusion and Suggestions

5.1 Conclusion

The Food Redistribution & Donation System provides a comprehensive, scalable, and socially driven digital solution to combat urban food waste and hunger. By offering a structured, role-based platform, the system effectively streamlines the entire donation lifecycle—from surplus food listing by donors to request handling by recipients, coordinated pickups, and automated notifications. This end-to-end orchestration supports operational efficiency and reinforces the platform's mission of sustainable resource redistribution.

This report has described the system structure based on simple software design concepts like modularity, abstraction, encapsulation, reusability, and separation of concerns. Moreover, the system adheres to prominent object-oriented design concepts, namely the SOLID principles, to ensure robustness, maintainability, and extensibility. To facilitate clean and scalable architecture, the architecture utilizes established software design patterns—Factory, Strategy, Observer, Singleton, and Decorator—each used with context-specific usage scenarios. The patterns facilitate component decoupling, service reuse, and dynamic behaviour extension across the system. The Donor, Recipient, and Admin role-based access model enhances usability, enforces privilege boundaries, and provides secure, role-based workflows. Significantly, the system also accommodates international initiatives towards sustainability by alignment with United Nations Sustainable Development Goals (SDGs) SDG 11: Sustainable Cities and Communities and SDG 12: Responsible Consumption and Production. Through assisting efficient surplus food redistribution, it not only pursues its technical and functional objectives but also provides a rich input towards social and environmental sustainability.

5.2 Suggestions for Future Enhancement

Although the current design forms a strong baseline, there are several opportunities for improvement to extend functionality and increase system adoption:

- **Mobile App Development**
  Introducing a cross-platform mobile app would offer users improved accessibility and real-time interaction, particularly useful for on-the-go donors and recipients.
- **Location-Based Optimization**
  Using geolocation and mapping services could improve logistics by optimizing pickup routes and enhancing location-sensitive food matching.
- **Volunteer Management Feature**
  Adding a module to manage community volunteers would strengthen the system's operational capacity, allowing volunteers to assist with transportation and outreach.
- **Intelligent Matching and Prediction**
  Implementing AI-based tools could enhance automated matching between donors and recipients, identify patterns in donation behaviour, and predict future needs.
- **Blockchain Integration**
  Using blockchain for logging donations and distributions can increase transparency, especially in partnerships with NGOs and public institutions.
- **Government and NGO Interoperability**
  Integrating with public welfare systems and food aid organizations via APIs would expand the system's reach and improve coordination across sectors.
- **Inclusive Design Features**
  Supporting multiple languages, speech-based input, and accessibility options would make the system more inclusive for users with diverse needs and backgrounds.

In conclusion, this SDS establishes a solid and well-architected foundation for the Food Redistribution & Donation System. With future enhancements and strategic collaborations, the platform can continue evolving into a smart, inclusive, and socially responsible digital infrastructure for sustainable food redistribution.

6.0 Bibliography/Reference

1. United Nations Environment Programme. (2021). *UNEP Food Waste Index Report 2021*. https://www.unep.org/resources/report/unep-food-waste-index-report-2021
2. Solid Waste Management and Public Cleansing Corporation (SWCorp). (2022, February 15). *Malaysia throws away 17,000 tonnes of food daily*. The Malaysian Reserve. https://themalaysianreserve.com/2022/02/15/malaysia-throws-away-17000-tonnes-of-food-daily/
3. Rivan, N. F. M., Yahya, H. M., Shahar, S., Singh, D. K. A., Ibrahim, N., Ludin, A. F. M., ... & Chan, Y. H. (2022). Changes in health-related lifestyles and food insecurity and its association with quality of life during the COVID-19 lockdown in Malaysia. *BMC Public Health*, 22, 1-10. https://doi.org/10.1186/s12889-022-13334-2
4. **Lim, S. H., Chin, N. L., & Yusof, A. Y.** (2024). *Comprehensive legislation development and policy framework for food waste management in Malaysia*. *Journal of Advanced Research Design*, 117(1), 1–10. https://akademiabaru.com/submit/index.php/ard/article/download/5324/4144/26819
5. Food and Agriculture Organization (FAO). (2022). *The State of Food and Agriculture 2022: Leveraging automation in agriculture for transforming agrifood systems*. Rome: FAO. Retrieved from https://www.fao.org/documents/card/en/c/cc2960en/
6. Feeding America. (2023). *Facts about food waste in America*. Retrieved from https://www.feedingamerica.org/about-us/press-room/food-waste
7. Food Rescue US. (n.d.). *Our Mission*. Retrieved May 2025, from https://foodrescue.us/
8. Food Rescue US. (n.d.). *How It Works*. Retrieved May 2025, from https://foodrescue.us/how-it-works/
9. Food Rescue US. (n.d.). *Impact Dashboard*. Retrieved May 2025, from https://foodrescue.us/impact/
10. Varghese, C., Pathak, D., & Varde, A. S. (2021). SeVa: A food donation app for smart living. *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*, 9375945. https://doi.org/10.1109/CCWC51732.2021.9375945
11. Nalawade, S., More, A., Zodge, A., & Hivarekar, M. (2024). A review paper on Annapurnata: An online food donation application. *International Journal for Research in Applied Science and Engineering Technology*, 12(2), 1101–1104. https://doi.org/10.22214/ijraset.2024.58318
12. Chaudhary, K. S. (2021). Economic impact of food waste in urban India. *Wikifarmer*. Retrieved from https://wikifarmer.com/library/en/article/economic-impact-of-food-waste-in-urban-india

13. Cecchini, M., & Warin, L. (2016). *Impact of food waste reduction initiatives on food security and sustainability* (OECD Food, Agriculture and Fisheries Papers, No. 92). OECD Publishing. https://doi.org/10.1787/5jlr3c6cr84n-en
14. OLIO. (2023). *Food Waste Heroes Programme*. https://olioapp.com