University of Belize

Report #3: Gated Community Application

Group #1

Kelsey A., Luigi A., Jordani A., Arthur B., Aiyesha C., Aiden P.

University of Belize

CMPS4131- Software Engineering

Mr. Medina Manuel

May 22, 2025

Table of Contents

Customer Statement of Requirements (CSR)

Problem Statement

Security Personnel's Perspective

Currently, security personnel face significant challenges with the manual visitor management system. Visitors are recorded by hand in logbooks, which leads to inefficiencies and potential errors. Each time a visitor arrives, their details must be manually written down, causing delays, especially during peak hours. This results in long queues and a slower check-in process, making it difficult for security personnel to manage the flow of visitors effectively.

A critical issue arises during identity verification. If a host is unavailable or unresponsive to phone calls, security personnel cannot easily verify if a visitor is authorized to enter. This lack of confirmation poses a security risk and creates delays. Even when the host is available, the reliance on manual approval processes wastes time and can lead to critical delays, particularly in urgent situations when quick decisions are needed.

Furthermore, the paper-based system is prone to human error. Inaccurate entries and unclear handwriting make it difficult to track visitors properly. Retrieving past records is time-consuming and inefficient, especially in situations where quick access to information is needed, such as in emergency response scenarios.

The security personnel also lack real-time access to a list of visitors on-site. This makes it challenging to determine whether a person is authorized to enter the building or if there is someone unauthorized on the premises. Without this instant access to visitor data, security becomes vulnerable to potential breaches. The situation worsens when security has to rely on hosts for entry approvals. If a host is unavailable or does not respond in time, the visitor is left waiting, and security is unable to make a timely decision, compromising both efficiency and security.

Security Staff Perspective

The manual system currently in place is outdated and inefficient, relying heavily on human effort for tasks that could be automated. Paper logbooks and the need for phone calls to hosts slow down the visitor entry process significantly. During peak hours, security personnel struggle to manage the visitor flow effectively, as the manual system cannot keep up with the volume of visitors.

Security staff are particularly challenged by the lack of real-time access to the list of visitors currently on-site. Without this immediate data, it is difficult for security personnel to verify if someone is authorized to enter or if there is a person on the premises who should not be there. The inability to access visitor data instantly creates a vulnerability that could potentially lead to security breaches. This issue is compounded when approval from hosts is required. The delays caused by waiting for host responses make security staff dependent on the hosts' availability, which can be unreliable.

Security personnel also find the process of retrieving past visitor records cumbersome. The paper-based logbooks make it time-consuming to look up previous records, which becomes especially problematic when quick access to past data is necessary in emergencies. This lack of efficient record retrieval hampers the security staff’s ability to act quickly and decisively in critical situations.

Visitor’s Perspective

Visitors, whether they are family members, delivery personnel, service providers, or other types of guests, often find the current system frustrating and time-consuming. First-time visitors are required to fill out lengthy registration forms, which adds unnecessary time to the check-in process. Even regular visitors are inconvenienced by the need to provide the same information every time they arrive, making the process feel redundant and inefficient.

One of the major frustrations is the waiting time when the host is unavailable to approve entry. If the host does not answer the phone, visitors are left uncertain about whether they will be allowed in, which adds stress and prolongs their wait. Visitors often find themselves in limbo, unsure of

the next steps, especially if they do not have prior approval. This uncertainty leads to confusion and inconvenience, creating a negative experience for the visitors.

Moreover, for visitors arriving without prior approval, the entire experience becomes even more chaotic. They are left unsure about whether they will be allowed in, leading to delays and confusion. The lack of a streamlined and predictable process makes it difficult for visitors to know what to expect, causing frustration and dissatisfaction.

To address the challenges outlined, we envision a system that will significantly enhance efficiency, reduce human error, and improve overall security. The proposed system will include pre-registration, allowing visitors to be registered before their arrival. This will eliminate the need for manual data entry each time a visitor arrives, speeding up the check-in process and particularly benefiting frequent visitors. The system will also provide security personnel with real-time access to data on visitors currently on-site, enabling immediate verification of whether a visitor is authorized to enter. Technologies such as QR code scanning or facial recognition can be implemented to facilitate quicker and more accurate identification of visitors, ensuring faster and safer entry.

With digital approval processes, hosts will be able to approve or deny visitor entry directly from their mobile devices, eliminating the need for phone calls and reducing approval delays. This will allow security personnel to handle approvals on the spot, speeding up the entry process and minimizing waiting times. The system will include real-time alerts for unauthorized entry attempts or potential security threats, allowing security staff to respond swiftly to any incidents. Additionally, digital visitor records will be stored securely, providing easy and efficient access to past visitor data, which will be crucial in emergency situations.

To ensure security, efficiency, and compliance, the visitor management system will implement key policies. Data protection will be a priority, safeguarding visitor information per regulations like GDPR. Authorization levels will restrict sensitive actions to security personnel and designated hosts, reducing unauthorized access. Visitors will be categorized (e.g., family, delivery personnel) for a smoother check-in process. These policies will enhance security, streamline operations, and improve the visitor experience.

Glossary of Terms

1. **Access Control –** The process of restricting entry to a facility based on predefined authorization levels.
2. **Authorization Levels –** System-defined access control measures that restrict specific actions (e.g., approving visitors, viewing confidential details) to authorized security personnel and designated hosts.
3. **Check-in Process** – The procedure visitors follow upon arrival, including identity verification, host approval, and registration.
4. **Compliance –** Adherence to regulations and policies (such as GDPR) to ensure the secure processing and storage of visitor data.
5. **Data Protection** – Measures implemented to safeguard visitor information from unauthorized access or breaches.
6. **Digital Approval Process –** A system that allows hosts to approve or deny visitor entry remotely via mobile devices, eliminating reliance on phone calls and reducing delays.
7. **Emergency Response –** The ability of security personnel to quickly retrieve visitor records and access real-time data to act promptly in urgent situations.
8. **Identity Verification –** The process of confirming a visitor's authorization to enter the premises, which may include QR code scanning, facial recognition, or host approval.
9. **Manual Visitor Management System –** The traditional paper-based method of recording visitor details in logbooks, which leads to inefficiencies, errors, and security risks.
10. **Pre-registration –** A feature that allows visitors to register before arrival, reducing manual data entry and expediting the check-in process.
11. **Queue Management** – The process of reducing visitor wait times by implementing efficient check-in procedures.
12. **Real-time Access –** The ability of security personnel to instantly view the list of visitors currently on-site for quick verification and decision-making.
13. **Security Breach –** An incident where an unauthorized individual gains access to a restricted area due to system vulnerabilities or procedural failures.

14. **Security Personnel –** Staff responsible for managing visitor entry, verifying identity, ensuring compliance, and responding to security threats.
15. **Security Threats –** Potential risks such as unauthorized entry attempts, unverified visitors, or delays in security processes that may compromise safety.
16. **Self-check-in** – A feature that allows visitors to complete the check-in process independently using a digital kiosk or mobile app.
17. **Streamlined Check-in Process –** A system that categorizes visitors (e.g., family, delivery personnel) for faster and more efficient entry, particularly benefiting frequent visitors.
18. **Unauthorized Entry Attempt** – When an individual without proper approval tries to access a restricted area.
19. **Visitor Categories –** Groups such as family members, delivery personnel, and service providers, each with tailored entry requirements to facilitate a smooth check-in process.
20. **Visitor Management System (VMS) –** A proposed digital system designed to improve visitor tracking, reduce errors, and enhance security by automating registration, identity verification, and approval processes.

System Requirements

Priority Weight	Description
1	Not Important
2	Low Importance
3	Normal
4	Important
5	Very Import

Table 1. System Requirements Priority Scale

Our input for the system requirements was based on research from best practices and recommendations for gated communities. We reviewed multiple sources to ensure that our features align with the security, convenience, and efficiency needs of gated communities (PalAmerican Security, 2021).

Enumerated Functional Requirements

REQ-X	Priority Weight	Description
REQ-1	1	The system shall allow all users (administrators, residents, and security guards) to log in using a username and password.
REQ-2	4	The system shall allow residents and security guards to communicate via notification system.
REQ-3	3	The system shall allow visitors to provide feedback after their visit.
REQ-4	4	The system shall allow residents and security to report visitors for blacklisting.
Admin		
REQ-5	5	The system shall allow administrators to approve blacklist requests.
REQ-6	4	The system shall allow administrators to view resident and security guard information.
REQ-7	4	The system shall allow administrators to add resident and security guard

		residents.
REQ-8	4	The system shall allow administrators to edit resident and security guard information.
REQ-9	5	The system shall allow administrators to delete resident and security guard records.
REQ-10	5	The system shall allow administrators to remove visitors from the blacklist if necessary.
Security Personnel		
REQ-11	4	The system shall allow security guards to view visitor history.
REQ-12	5	The system shall allow security guards to scan QR codes.
REQ-13	4	The system shall allow security guards to scan the next QR code as people enter.
REQ-14	4	The system shall allow security guards to verify recurring visitor QR codes against the numbers of attendees.
REQ-15	5	The system shall allow security guards to log any incidents or concerns about visitors.
Residence		
REQ-16	4	The system shall allow residents to view and edit visitor information.
REQ-17	4	The system shall allow residents to add visitor details.
REQ-18	5	The system shall allow residents to generate QR codes for visitors.
REQ-19	5	The system shall allow residents to send QR codes to visitors.
REQ-20	4	The system shall allow residents to register recurring visitors with predefined schedules.
REQ-21	4	The system shall allow residents to modify or revoke access for recurring visitors at any time.
REQ-22	4	The system shall allow residents to generate one QR code for multiple event guests.

REQ-23	3	The system shall allow residents to rate visitor experiences.

Table 2. Functional Requirements

Enumerated Nonfunctional Requirements

NONREQ-X	Priority Weight	Description	FURPS+
NONREQ-1	1	The system shall lock an account for 30 minutes after five (5) failed login attempts.	Security
NONREQ-2	4	The system shall ensure that only authorized residents can generate and send QR codes.	Security
NONREQ-3	5	The system shall log all user activities, including resident updates and security guard scans.	Security
NONREQ-4	1	The system should only allow one instance of a login for an account.	Security
NONREQ-5	5	The system shall ensure that all data is kept secure to protect residents' and visitors' information.	Security
NONREQ-6	2	The system shall work smoothly on both mobile and desktop devices.	Performance
NONREQ-7	5	The system shall allow access only if the entered credentials match those in the database.	Security
NONREQ-8	4	The system shall have an intuitive user interface (UI) for easy navigation across all user roles.	Usability
NONREQ-9	5	The system shall allow QR code scanning only if the visitor is registered and meets access conditions.	Security
NONREQ-10	3	The system shall be available at least 99% of the time to ensure users can always access it.	Reliability
NONREQ-11	3	The system shall provide a "Remember Me" option for user convenience to login (excluding security guards).	Usability
NONREQ-12	5	The system shall deliver messages or alerts between residents and security guards within 5 seconds to ensure timely communication.	Reliability
NONREQ-13	5	The system shall ensure QR codes for recurring	Reliability

		visitors expire after their scheduled time.	
NONREQ-14	3	The system shall notify residents when their recurring visitor successfully checks in or is denied entry.	Reliability
NONREQ-15	3	The system shall provide clear error messages when guests attempt to use an expired or invalid QR code.	Usability
NONREQ-16	3	The system shall ensure visitor feedback is submitted anonymously unless the user chooses otherwise.	Security
NONREQ-17	3	The system shall allow authorized users (residents, security, admins) to review and respond to feedback.	Usability
NONREQ-18	4	The system shall provide real-time alerts when a blacklisted visitor attempts to enter.	Security
NONREQ-19	5	The system shall ensure only authorized personnel can modify the blacklist.	Security

Table 3. Nonfunctional Requirements

On-Screen Appearance Requirements

ONSREQ-X	Priority Weight	Description
ONSREQ-1	4	Confirmation messages (e.g., "QR Code Scanned Successfully") shall be displayed in green for clarity.
ONSREQ-2	2	The system shall show a loading spinner when processing login, QR code scans, or data retrieval.
ONSREQ-3	5	The system must provide clear navigation options for users to easily access their assigned features.
ONSREQ-4	5	The Admin's and Security’s screen must be actively updated to show the latest resident and visitor logs.
ONSREQ-5	5	The Resident's screen must display visitor status updates, including pending and approved QR codes.
ONSREQ-6	2	The system must provide clear error messages after each user action.

Table 4. On-Screen Requirements

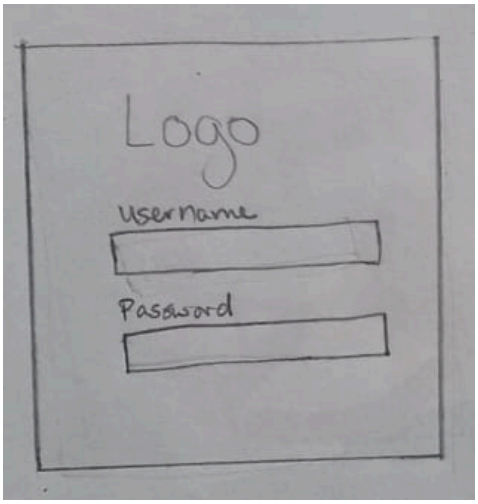

Fig 1. Login Screen

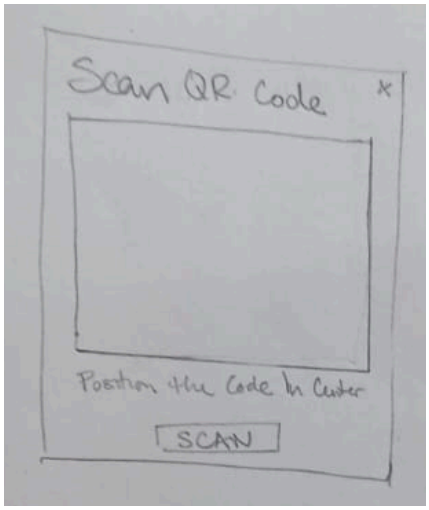

Fig 2. Scan QR Code

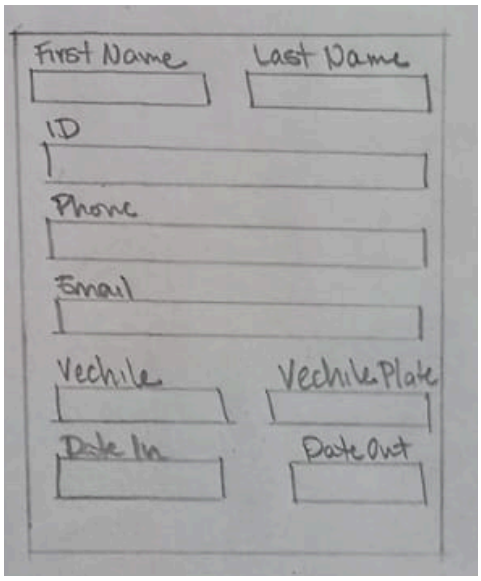

Fig 3. Adding A Visitor

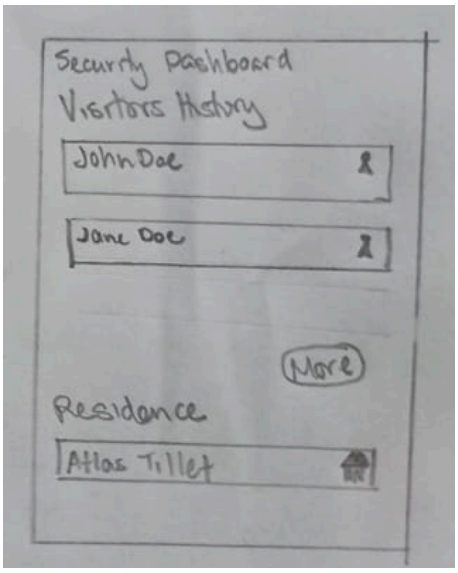

Fig 4. Security Dashboard

Our app features are inspired by Hope, Tzib and Shol's first prototype (2024). Some features will also be incorporated along with our team's new ideas.

Functional Requirements Specification

Stakeholders

Residents	Visitors	Security Guards
Administrators	Homeowners' Association	

Table 5. Stakeholders of the System

Actors

Actor	Roles	Type	Goals
Residents	● Manage visitor access by generating and sending QR codes, ● Registering recurring visitors, ● Reporting visitors for blacklisting, and ● Rating visitor experiences.	Initiating	● Ensure secure, controlled access for their visitors. ● Streamline visitor management and maintain oversight over guest activity. ● Provide feedback to improve system functionality and security.
Visitors	● The end-users who present QR codes to gain entry and provide feedback after their visit.	Initiating	● Gain authorized and hassle-free entry into the gated community. ● Communicate their visit experience for potential improvements.
Security Guards	● Verify visitor entries by scanning QR codes, ● Checking recurring visitor schedules, ● Logging incidents, and ● Monitoring the blacklist status.	Participating	● Maintain the safety and security of the community by ensuring only authorized visitors are admitted. ● Accurately log and report any incidents or anomalies.
Administrators	● Oversee system operations, ● Manage resident	Initiating	● Ensure the system operates securely, efficiently, and reliably.

	records, ● Approve or reject blacklist requests ● Remove visitors from the blacklist when necessary.		● Maintain up-to-date and accurate records for residents and visitors. ● Enforce community policies and ensure compliance with security standards.

Table 6. Actor Types and Goals

Use Cases

Casual Description

UC Name	Actor	Actor's Goal	Req #
UC-1 Login	Resident/ Security Guard/ Administrator	To access the system using their login credentials.	REQ-1 NONREQ-1 NONREQ-4 NONREQ-7 NONREQ-11
UC-2 Create Visitor Schedule	Resident	To register a visit with a predefined schedule.	REQ-16 REQ-17 REQ-20 REQ-21 REQ-22 NONREQ-8 NONREQ-10 NONREQ-13
UC-3 Generate QR Code	Resident	To generate a QR code for an expected visitor.	REQ-18 REQ-22 NONREQ-2 NONREQ-8 NONREQ-10
UC-4 Send QRCode	Resident	To send a generated QR code to a visitor for entry.	REQ-19 NONREQ-2 NONREQ-8

			NONREQ-10
UC-5 Send Notifications	System	To be informed when a visitor successfully checks in or is denied entry.	REQ-2 NONREQ-10 NONREQ-12 NONREQ-14
UC-6 Submit Feedback	Visitor	To provide feedback on their visit.	REQ-3 REQ-17 REQ-23 NONREQ-15 NONREQ-16 NONREQ-17
UC-7 Scan QR	Security Guard	To scan a visitor's QR code and allow or deny entry based on access conditions.	REQ-12 REQ-13 REQ-14 NONREQ-6 NONREQ-8 NONREQ-9 NONREQ-13 NONREQ-14 NONREQ-15
UC-8 View History	Security Guard	To review past visitor entries.	REQ-11 NONREQ-3 NONREQ-8 NONREQ-14
UC-9 Manage Blacklist	Resident/ Security Guard	To report a visitor for an act that is not acceptable by the community or a specific resident.	REQ-4 REQ-15 NONREQ-8 NONREQ-18 NONREQ-19
UC-10 Add to Blacklist	Administrator/ Resident	To approve a blacklist request submitted by residents or security.	REQ-5 NONREQ-8 NONREQ-19
UC-11 Remove	Administrator	To remove a visitor from the blacklist	REQ-10

from Blacklist		when necessary.	NONREQ-8 NONREQ-19
UC-12 ManageUsers	Administrator	Manage user accounts, including creating new accounts, modifying account details, or deleting accounts.	REQ-6 REQ-7 REQ-8 REQ-9 NONREQ-5 NONREQ-8
UC-13 AddUser	Administrator	To register a new user in the system.	REQ-7 NONREQ-5 NONREQ-8
UC-14 EditUser	Administrator	To manage user details by editing records.	REQ-6 REQ-8 NONREQ-5 NONREQ-8
UC-15 Delete User	Administrator	To manage user details by deleting records.	REQ-6 REQ-9 NONREQ-5 NONREQ-8
UC-16 View Feedback	Administrator/ Resident/ Security Guard	To view feedback submitted by visitors.	REQ-3 REQ-23 NONREQ-8 NONREQ-16 NONREQ-17

Table 7. Brief Description of Use Case

Use Case Diagram

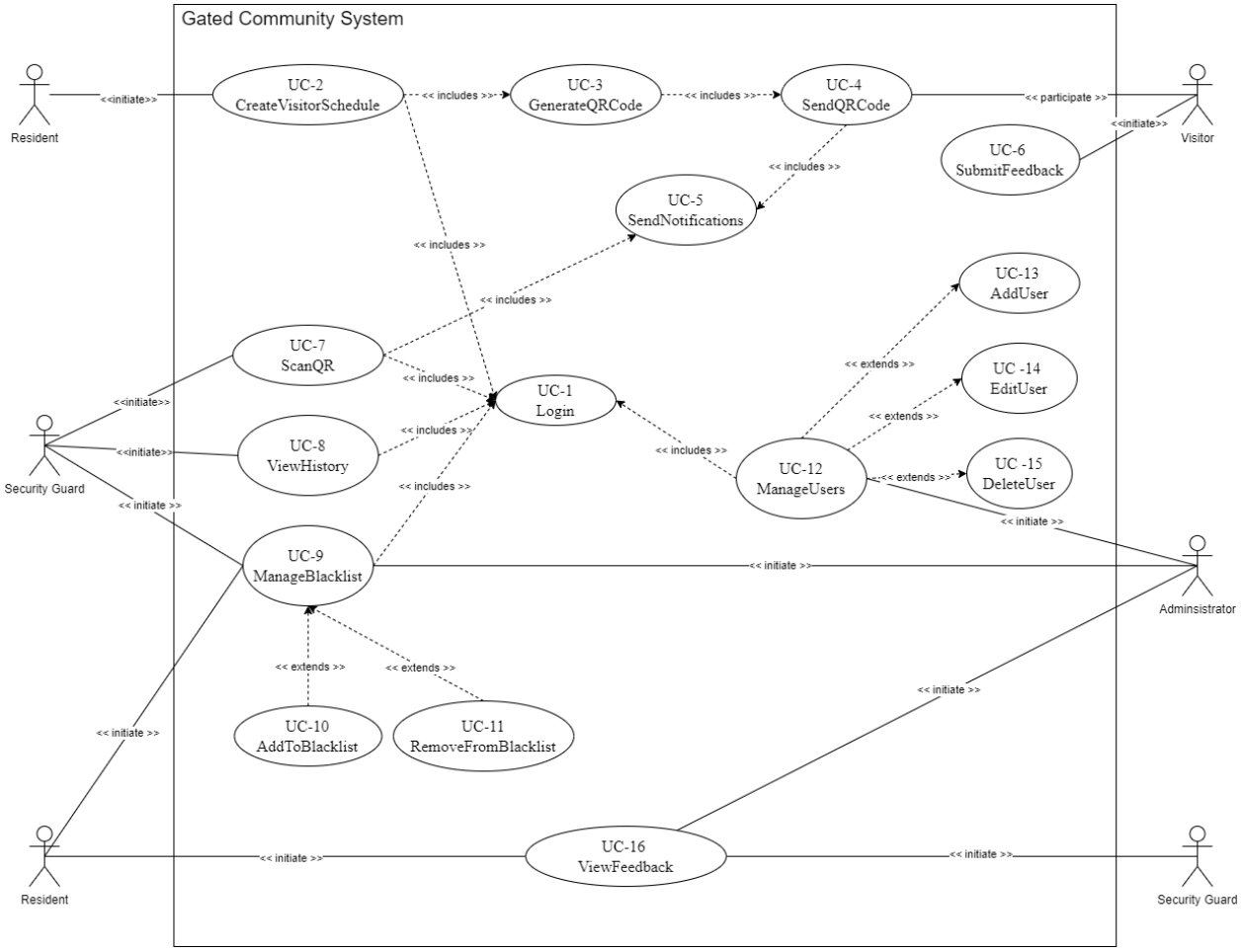

Fig 5. Use Case Diagram for the Gated Community System

Traceability Matrix

Use Cases		UC-1	UC-2	UC-3	UC-4	UC-5	UC-6	UC-7	UC-8	UC-9	UC-10	UC-11	UC-12	UC-13	UC-14	UC-15	UC-16
PW	Reqs																
1	REQ-1	X															
4	REQ-2					X											
3	REQ-3						X										X
4	REQ-4									X							
5	REQ-5										X						
4	REQ-6												X		X	X	
4	REQ-7												X	X			
4	REQ-8										X		X		X		
5	REQ-9												X			X	
5	REQ-10											X					
4	REQ-11								X								
5	REQ-12							X									
4	REQ-13							X									
4	REQ-14							X									
5	REQ-15									X							
4	REQ-16		X														
4	REQ-17		X				X										
5	REQ-18			X													
5	REQ-19				X						X						
4	REQ-20		X														
4	REQ-21		X														
4	REQ-22		X	X													
3	REQ-23						X										X
1	NONREQ-1	X															

4	NONREQ-2				X												
5	NONREQ-3								X								
1	NONREQ-4	X															
5	NONREQ-5												X	X	X	X	
2	NONREQ-6							X					X				
5	NONREQ-7	X															
4	NONREQ-8		X					X	X	X		X	X	X			X
5	NONREQ-9							X									
3	NONREQ-10		X	X	X	X											
3	NONREQ-11	X															
5	NONREQ-12					X											
5	NONREQ-13		X					X									
3	NONREQ-14					X		X	X								
3	NONREQ-15						X	X									
3	NONREQ-16						X										X
3	NONREQ-17						X										X
4	NONREQ-18									X							
5	NONREQ-19									X		X					
	Max PW	1	5	3	3	1	5	5	5	5	3	3	5	5	2	3	5
	Total PW	11	32	12	12	15	19	35	16	22	14	14	28	13	13	14	16

Table 8. Requirements Mapped to Use Cases

Fully-Dressed Description

Use Case UC-2	**CreateVisitorSchedule**
Related Requirements:	Functional Requirements: REQ-16, REQ-17REQ-20, REQ-21, REQ-22 Non-Functional Requirements: NONREQ-8,NON-REQ-10 NONREQ-13
Initiating Actor(s):	Resident
Actor's Goal:	Successfully register a visit session in the system with a predefined schedule.
Participating Actor:	Visitor
Preconditions:	● The resident must be logged into the system.
Post conditions:	● The visitor is successfully registered in the system ● The QR code was generated and sent to the visitor
Flow of Events for Main Success Scenario:	
→	1. Resident navigates to the visitor registration page.
	2. Resident enters valid visitor details, including visit frequency and duration.
←	3. System generates a persistent QR code for the visitor: Include UC-3 Generate QR Code
	4. System notifies the security team about the new recurring visitor: Include UC-5 SendNotifications and Include UC-4 SendQRCode
	5. Visitor details are stored in the database for future visits.
Flow of Events for Extensions (Alternate Scenarios):	
	2. The user enters invalid information for the user. a. The system notifies the user with an error message.

Table 9. Fully Dressed Description of CreateVisit

Use Case UC-6	**SubmitFeedback**
Related Requirements:	Functional Requirements: REQ-3, REQ-17, REQ-23 Non-Functional Requirements: NONREQ-15, NONREQ-16, NONREQ-17
Initiating Actor(s):	Visitor
Actor's Goal:	To provide feedback on their visit.
Participating Actor:	Database
Preconditions:	● Systems must have stored visitor check-in data
Post conditions:	● The feedback is successfully submitted and stored in the system.
Flow of Events for Main Success Scenario:	
→	1. Visitor scans the QR code.
←	2. The system retrieves the visitor's details and sends an email containing a feedback link.
→	3. The visitor opens the email and clicks the feedback link.
	4. The visitor is (a)redirected to the link, (b) fills out the feedback form and (c) submits it.
←	5. The system processes and stores the feedback.
	6. The system displays a confirmation message to the visitor.

Table 10. Fully Dressed Description of SubmitFeedback

Use Case UC-7	**ScanQR**
Related Requirements:	Functional Requirements: REQ-12, REQ-13, REQ-14 Non-Functional Requirements: NONREQ-6, NONREQ-8, NONREQ-9, NONREQ-13, NONREQ-14, NONREQ-15
Initiating Actor(s):	Security Guard
Actor's Goal:	Successfully scan a visitor's QR code and allow or deny entry based on access conditions.
Participating Actor:	Visitor
Preconditions:	● Visitors must be registered in the system with a valid QR code. ● Security guards must have access to the check-in system.
Post conditions:	The visitor ● Checks in ● Entry is granted, and ● The system logs the event.
Flow of Events for Main Success Scenario:	
→	1. Security navigates to the visitor scanning page.
	2. (a)Visitor arrives at the security checkpoint with QRcode and the (b) security guard scans the visitor's QR code using the check-in system
←	3. System validates the QR code against registered visitors.
→	4. Visitors are granted entry.
Flow of Events for Extensions (Alternate Scenarios):	
→	3. Visitor arrives at the security checkpoint with QRcode and the security guard scans the visitor's QR code using the check-in system. The QRcode was invalid. a. The visitor is not granted access.
←	4. The resident is notified of the potential visitor.

Table 11. Fully Dressed Description of ScanQR

Use Case UC-8	**ViewHistory**
Related Requirements:	Functional Requirements: REQ-11 Non-Functional Requirements: NONREQ-3, NONREQ-8, NONREQ-14
Initiating Actor(s):	Security Guard
Actor's Goal:	Review past visitor entries
Participating Actor:	Database
Preconditions:	● Security Guard must be logged into the system ● Systems must have stored visitor check-in data
Post conditions:	The visitor history is displayed.
Flow of Events for Main Success Scenario:	
→	1. Security guard navigates to the "Visitor History" section in the system
←	2. System displays a search interface.
→	3. Security guards enter filter information.
←	4. System (a) retrieves search information and (b) displays the relevant visitor history.
	5. Security guard reviews the records.
Flow of Events for Extensions (Alternate Scenarios):	
←	4. System returns (a) no data because information does not exist or (b) system fails to retrieve data.
	5. Security Guard receives a note that states if (a) there is no existing record or (b) the database is having specific issues.

Table 12. Fully Dressed Description of ViewHistory

Use Case UC-9	**ManageBlacklist**
Related Requirements:	Functional Requirements: REQ-4, REQ-15 Non-Functional Requirements: NONREQ-8, NONREQ-18, NONREQ-19
Initiating Actor(s):	Resident or Security Guard
Actor's Goal:	To report a visitor for an act that is not acceptable by the community or a specific resident.
Participating Actor:	Database
Preconditions:	● The resident or security guard must be logged into the system. ● Visitors must have a record in the system.
Post conditions:	The visitor ● Is added to the blacklist, preventing future check-ins or ● Is removed from the blacklist, allowing access
Flow of Events for Main Success Scenario:	
→	1. Resident or Security guard navigate to the "Manage Blacklist" section
←	2. System displays the list of visitors and search options
→	3. User enters a name or ID to search.
←	4. The system retrieves and displays the visitor's profile.
→	5. Resident or security guard selects "Add to Blacklist" or "Remove from Blacklist"
←	6. The system updates the visitor's access.
Flow of Events for Extensions (Alternate Scenarios):	
5a. Selected activity entails manage blacklist: Include UC-11 AddToBlacklist	
5b. Selected activity entails manage blacklist: Include UC-12 RemoveFromBlacklist	
←	4. System returns (a) no data or (b) system fails to retrieve data.
	5. Security Guard or Resident receives a note that states if (a) there is no existing record or (b) the database is having specific issues.

Table 13. Fully Dressed Description of ManageBlacklist

Use Case UC-12	**ManageUsers**
Related Requirements:	Functional Requirements: REQ-4, REQ-15 Non-Functional Requirements: NONREQ-8, NONREQ-18, NONREQ-19
Initiating Actor(s):	Administrator
Actor's Goal:	Manage user accounts, including creating new accounts, modifying account details, or deleting accounts.
Participating Actor:	Database
Preconditions:	● The administrator must be logged into the system. ● Usersmust have a record in the system.
Post conditions:	A user is added, edited or deleted from the system.
Flow of Events for Main Success Scenario:	
→	1. Administrator navigate to the "Manage Users" section
	2. Administrator selects the desired action "Add User," "Edit User", or "Delete User"
←	3. The system displays the relevant interface for (a) adding a user where a form is provided to enter user details or a (b) a search field to select a user for (b1) editing a user or (b2) deleting a user.
→	4. The admin provides the necessary input and submits the request
←	5. The system validates the input and processes the request.
	6. A confirmation message is displayed, indicating that the action was successfully completed.
Flow of Events for Extensions (Alternate Scenarios):	
3a. Selected activity entails manage users: Include UC-13 AddUser	
3b. Selected activity entails manage users: Include UC-14 EditUser	
3c. Selected activity entails manage users: Include UC-15 DeleteUser	
←	4. Required fields are missing when adding or editing a user: a. The system displays an error message stating "Please fill out all required fields before submitting"

Table 14. Fully Dressed Description of ManageUsers

Use Case UC-16	**View Feedback**
Related Requirements:	Functional Requirements: REQ-3, REQ-23 Non-Functional Requirements: NONREQ-8, NONREQ-16, NONREQ-17
Initiating Actor(s):	Administrator, Resident, or Security Guard
Actor's Goal:	To view visitor feedback submitted through the system.
Participating Actor:	Database
Preconditions:	● The Administrator, Resident, or Security Guard must be logged into the system. ● At least one feedback entry must exist in the system.
Post conditions:	The Administrator, Resident, or Security Guard successfully view visitor feedback.
Flow of Events for Main Success Scenario:	
→	1. Administrator, Resident, or Security Guard navigates to the "Visitor Feedback" section.
←	2. System retrieves and displays visitor feedback.
→	3. Administrator, Resident, or Security Guard view feedback.
→	4. The Administrator, Resident, or Security Guard exits the feedback.
Flow of Events for Extensions (Alternate Scenarios):	
←	2. System notifies the Administrator, Resident, or Security Guard with a message: "No feedback available at the moment."
→	3. Administrator, Resident, or Security Guard chooses to either (a) exit the section or (b) retry fetching feedback later.

Table 15. Fully Dressed Description of ViewFeedback

Sequence Diagrams

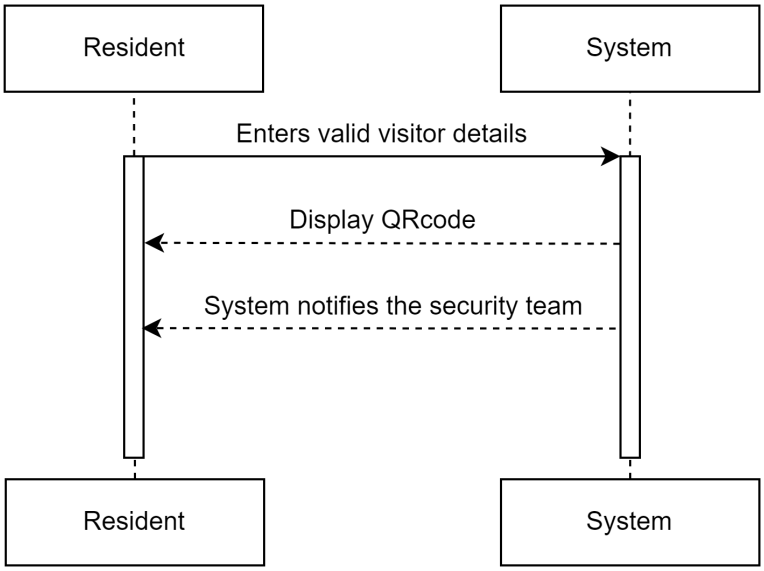

Fig 6. UC-2 CreateVisitorSchedule

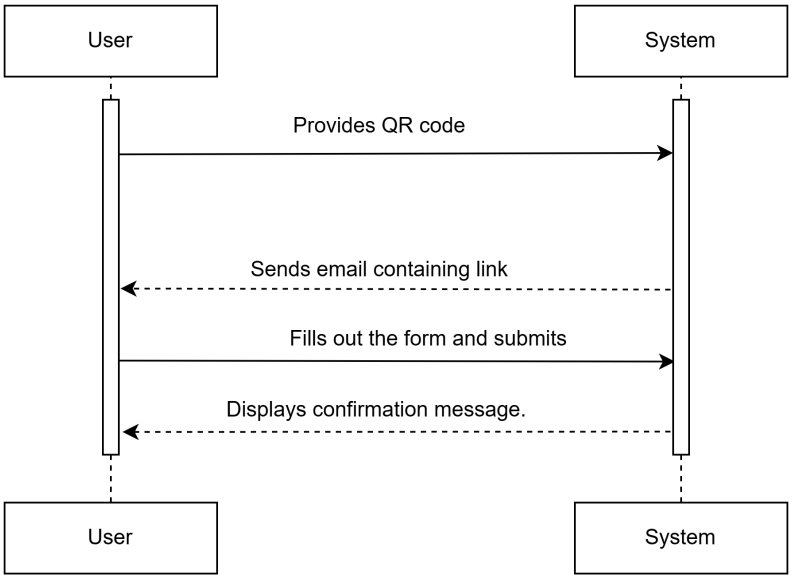

Fig 7. UC-6 SubmitFeedback

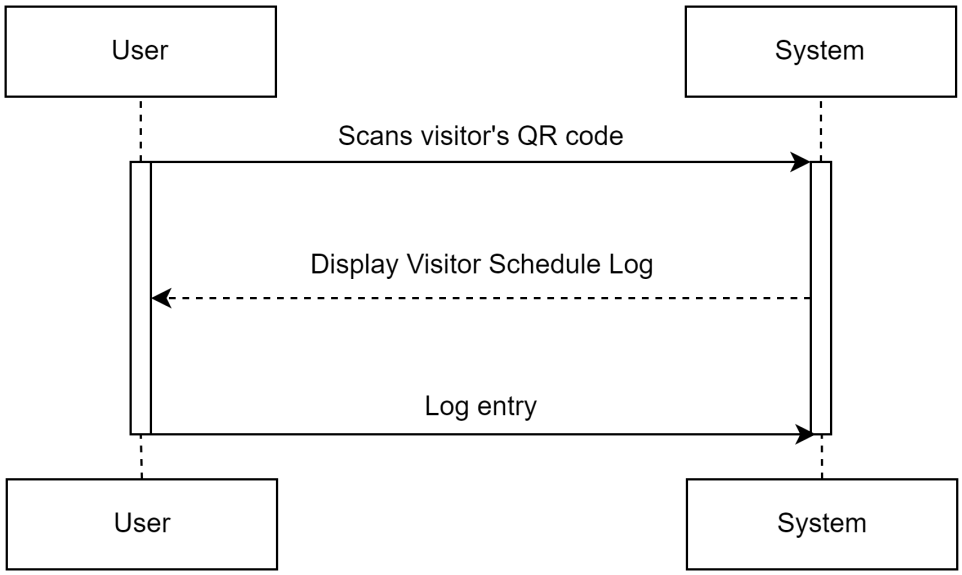

Fig 8. UC-7 ScanQR

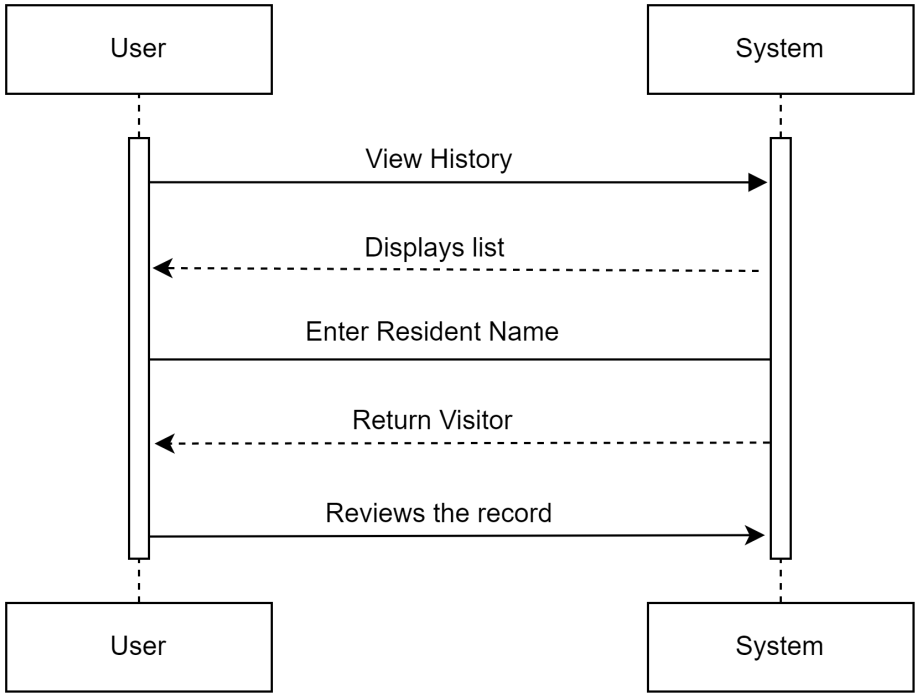

Fig 9. UC-8 ViewHistory

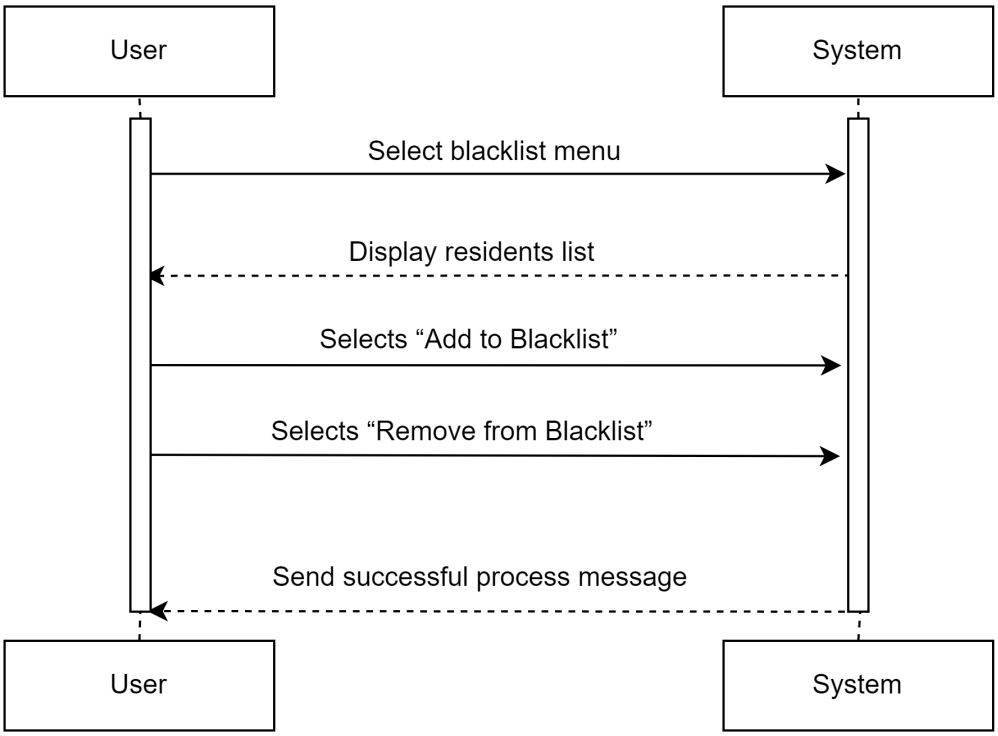

Fig 10. UC-9 ManageBlacklist

To effectively manage users, each process has a distinct flow, requiring a separate sequence diagram to accurately represent each one.

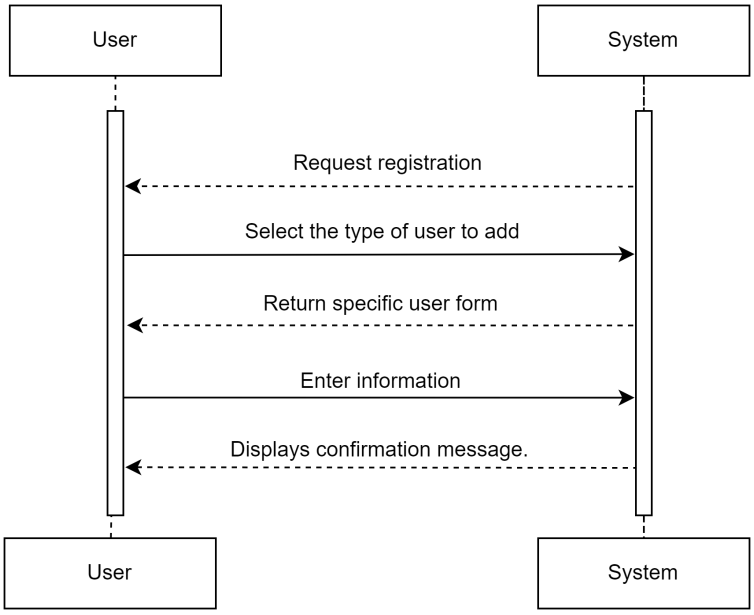

Fig 11. UC-13 AddUsers

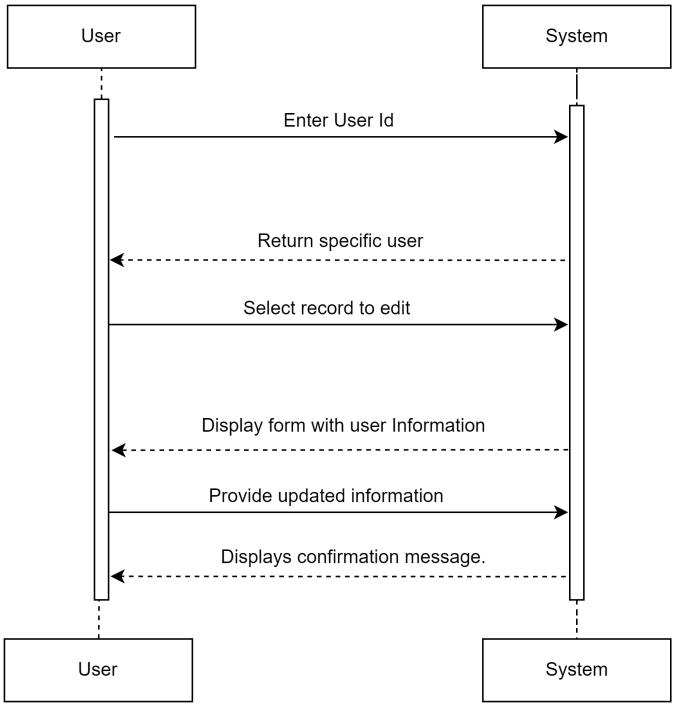

Fig 12.UC-14 EditUser

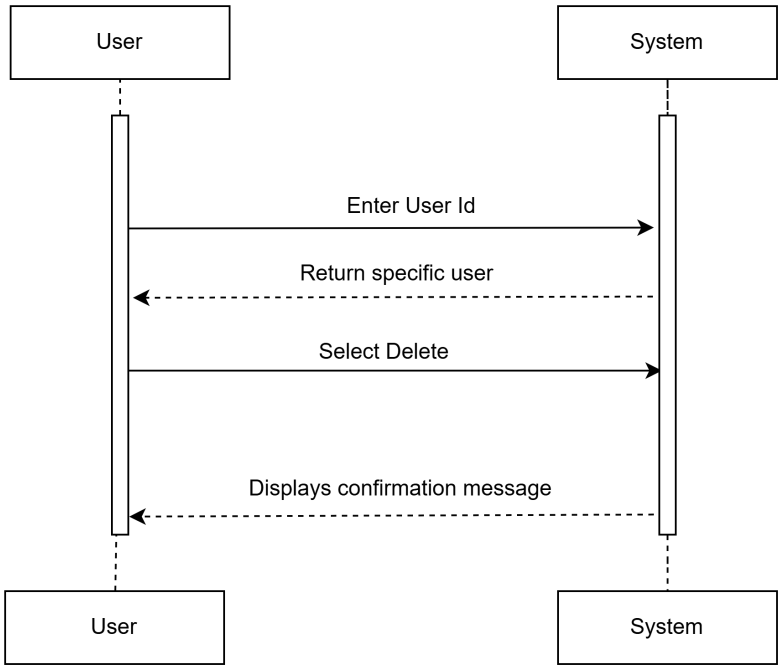

Fig 13.UC-15 DeleteUser

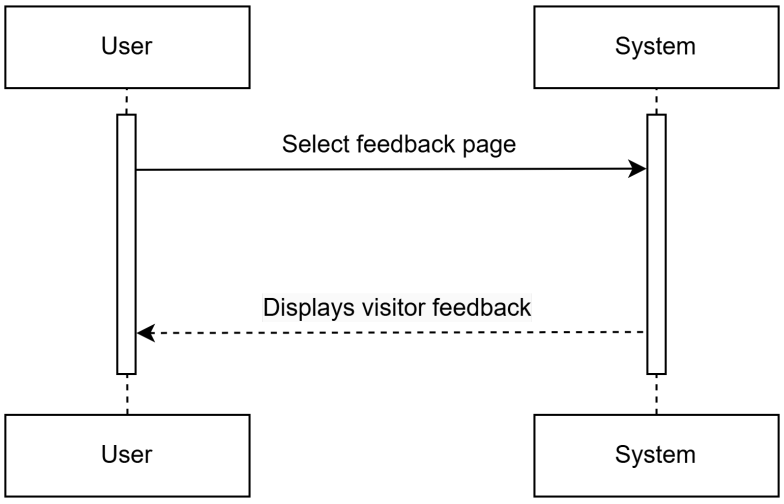

Fig 14. UC-16 ViewFeedback

User Interface Specifications

Preliminary Design

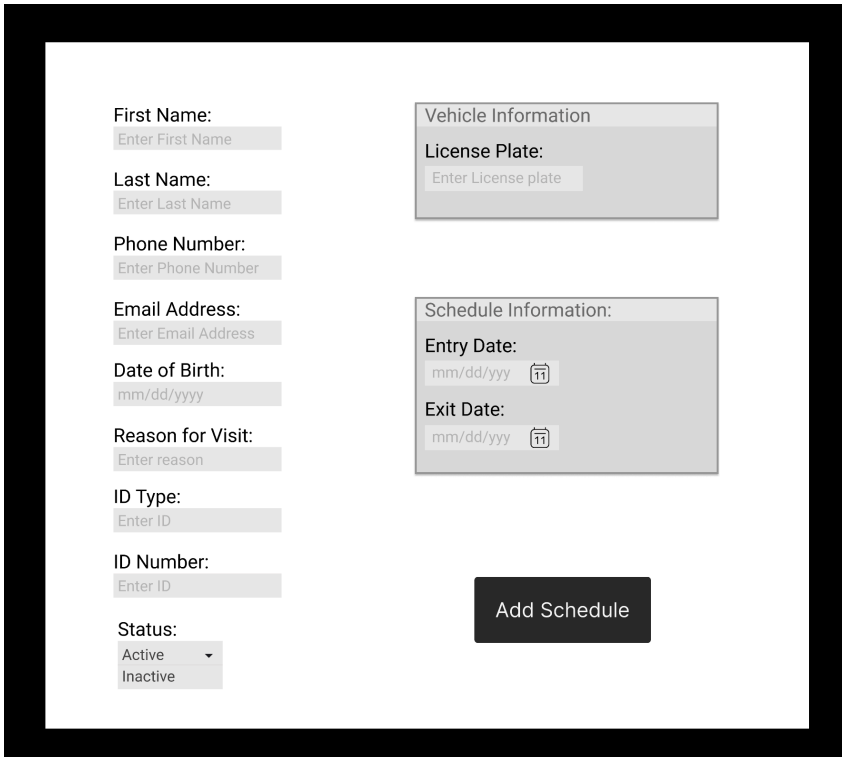

Fig 15. UC-2 CreateVisitor Schedule

To create a visitor schedule, the user must first fill in visitor information, vehicle details, and schedule information before clicking the 'Add Schedule' button. The visitor can be either active or inactive.

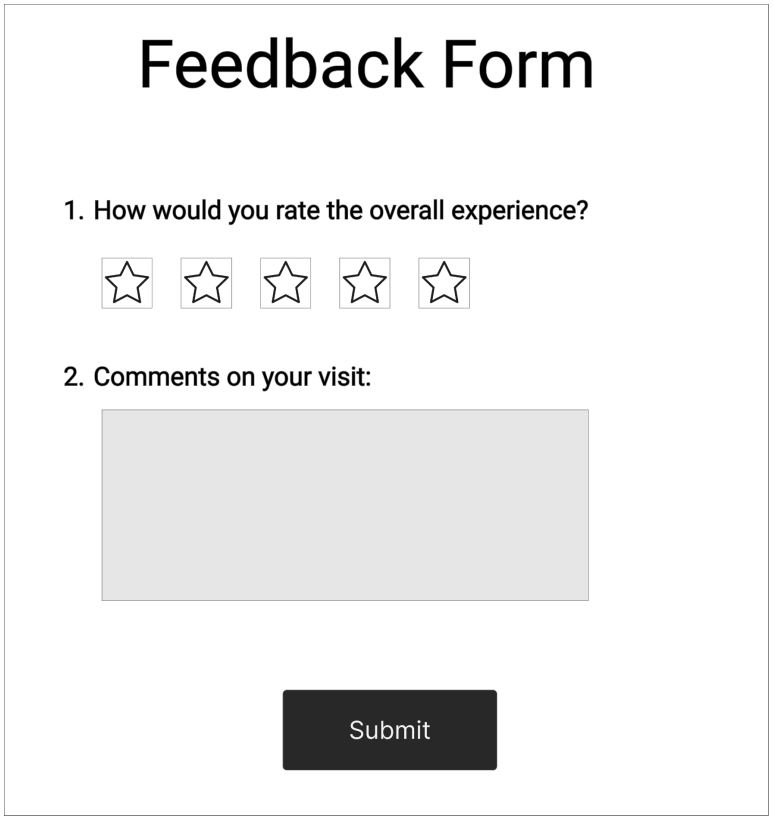

Fig 16. UC-6 SubmitFeedback

To submit a feedback form, the visitor will provide feedback using the star rating and can enter comments if desired.

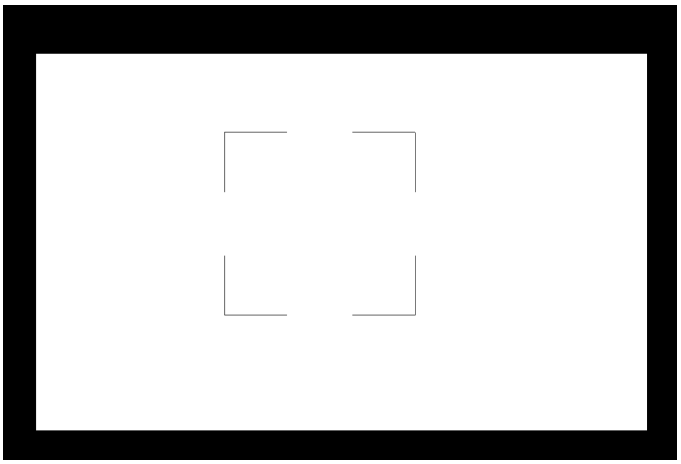

Fig 17. UC-7 ScanQR

To scan the QR code, the security guard will open the scanning page and scan the visitor's QR code to successfully grant entry.

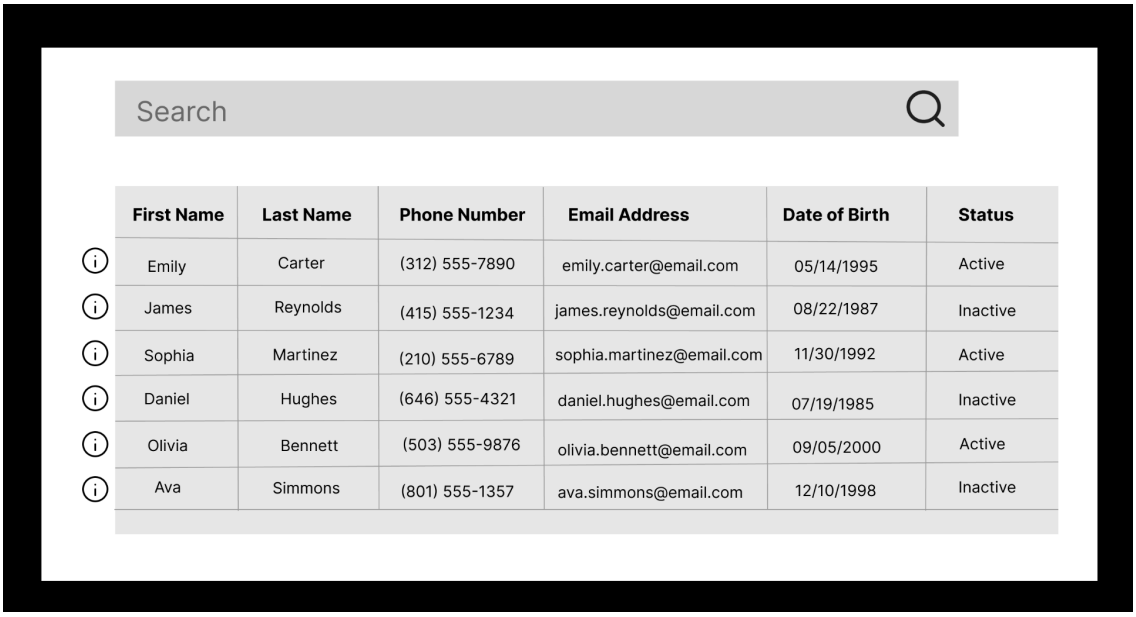

Fig 18. UC-8 ViewHistory

To view history, the user will navigate to the 'View History' page, where a list of visitors will be displayed. The user can also use the search bar to find specific visitors.Additionally, there is an information icon, and when clicked, it will display more details about the visitor.

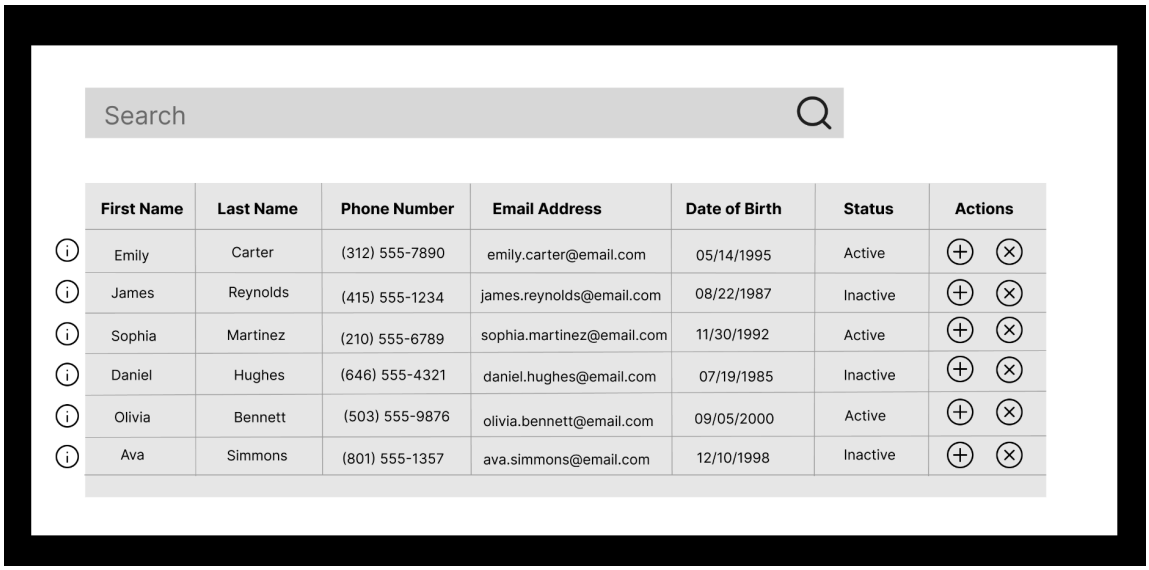

Fig 19. UC-9 ManageBlacklist

To manage the blacklist, the user will be displayed a table of all visitors with an actions column. They can add or remove a visitor from the blacklist by clicking the respective icons, which will trigger a confirmation popup.

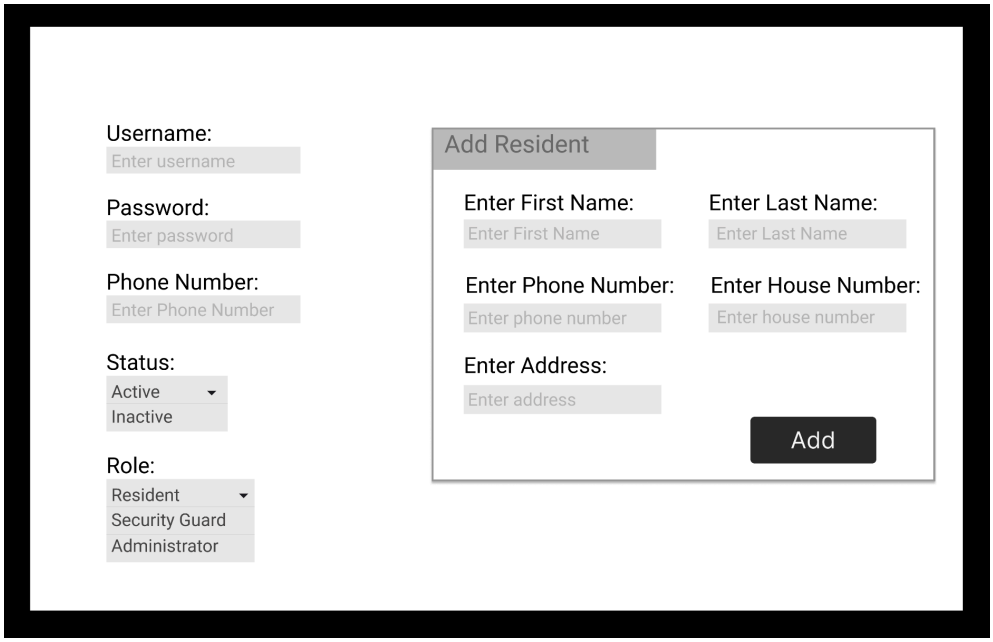

Fig 20. UC-13 AddUsers - Resident

To add a user, the admin must fill in all required fields. If the selected role is 'Resident,' a form will pop up on the side requesting resident information. The admin will enter the details and click the 'Add' button to successfully add the resident.

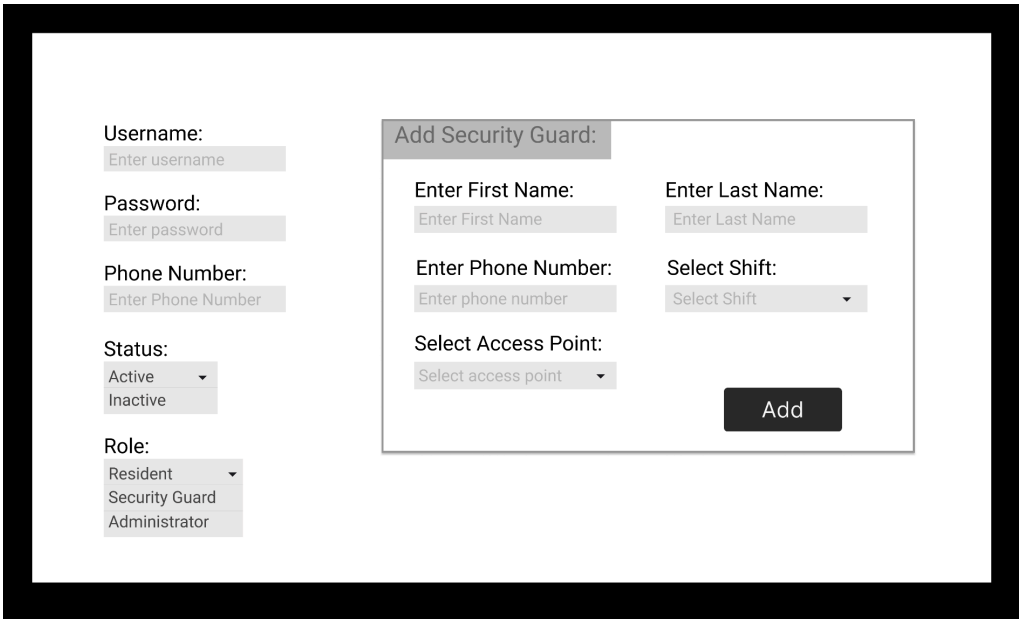

Fig 21. UC-13 AddUsers - Security Guard

The same process applies when adding a security guard. The admin must fill in the required information, and if 'Security Guard' is selected as the role, a form will pop up requesting security guard details. The admin will then enter the information and click the 'Add' button to successfully add the security guard.

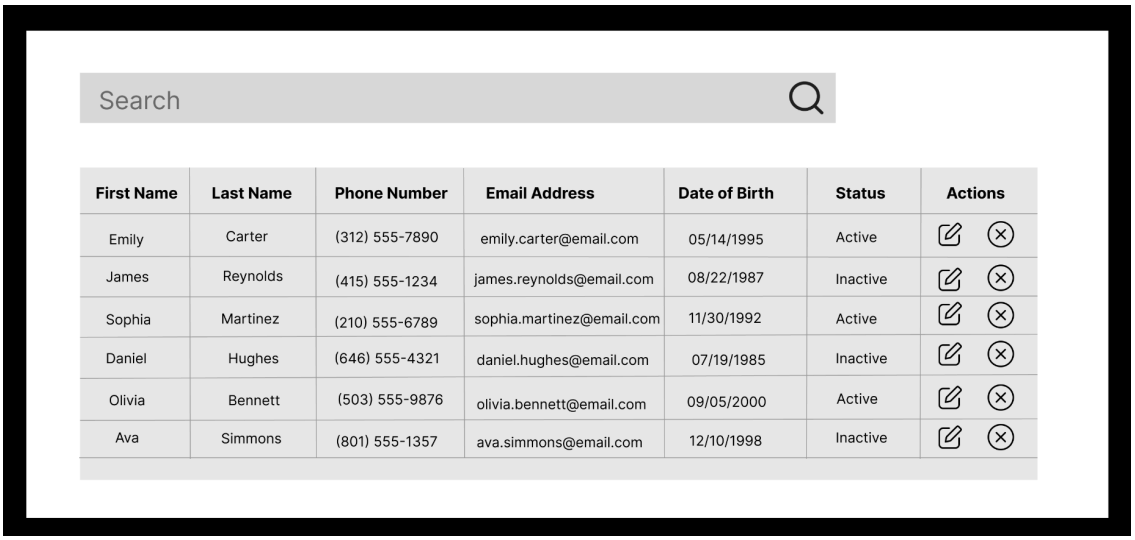

Fig 22. UC-14 EditUser and UC-5 Delete User

To edit or delete a user, the admin will open the 'List of Users' page, which will display a table with the list of users. The table includes an actions column with icons to edit or delete a user. The admin can choose to either edit or delete a user by clicking on the corresponding icon. If editing, they will be redirected to a page with the user's details to make changes. If deleting, a confirmation box will appear.

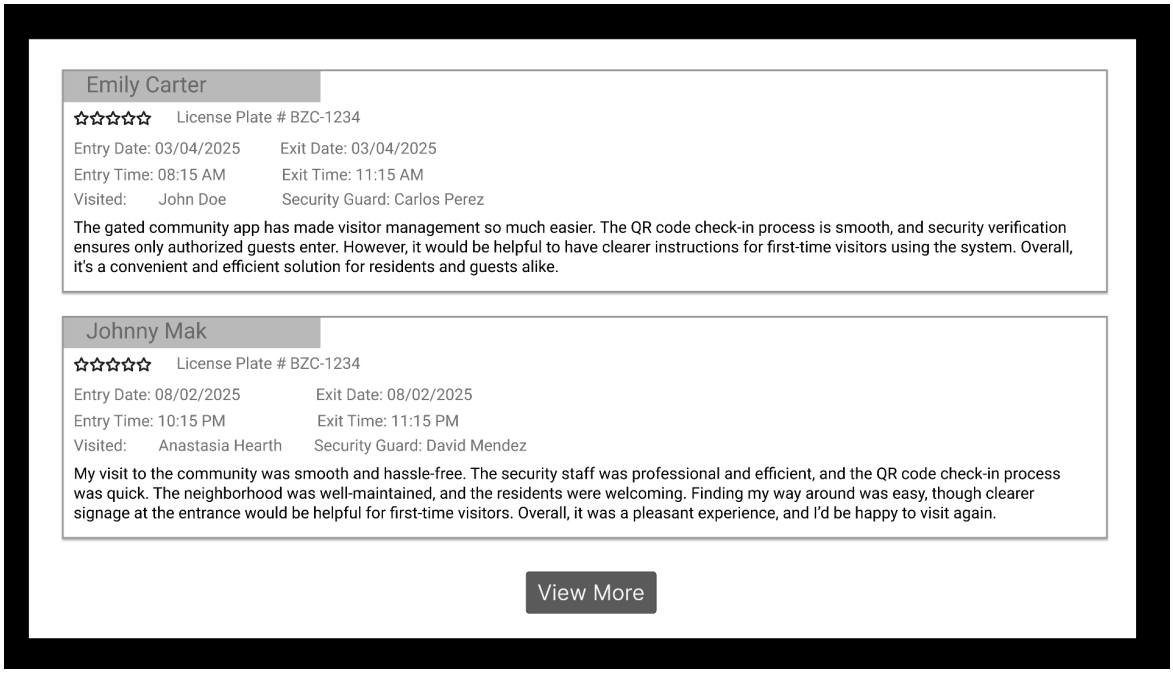

Fig 23. UC-16 View Feedback

To view visitor feedback, the user will go to the 'View Feedback' page, where a list of feedback will be displayed. They can click the 'View More' button to see additional visitor feedback.

Effort Estimation

User

Create Visitor Schedule (UC-2) - total 14 mouse clicks

1. Click "Schedule Visit" on the Home page.
2. Click on First Name Field
3. Click on Last Name Field
4. Click on Phone Number field
5. Click on Email Field
6. Click on DOB field
7. Click on Reason for Visit
8. Click on ID type Field
9. Click on ID Number Field
10. Select Status
11. Click on the licence plate field.
12. Click on the entry date field.
13. Click on the exit date field.
14. Click "Add Schedule."

Data Entry – 13 mouse clicks & 100 + keystrokes

1. Click in the First Name field and type First Name.
2. Click in the Last Name field and type Last Name.
3. Click in the Phone Number field and type Phone Number .
4. Click in the Email field and type Email .
5. Click in the DOB field and type Date of Birth.

6. Click in the Reason for Visit field and type Reason.
7. Click in the ID Type field and select ID Type (Dropdown).
8. Click in the ID Number field and type ID Number.
9. Click in the Status field and select Status (Dropdown).
10. Click in the License Plate field and type License Plate .
11. Click in the Entry Date field and type Entry Date.
12. Click in the Exit Date field and type Exit Date.
13. Click "Add Schedule."

Submit Feedback (UC-6) – total 5 mouse clicks

1. Click on the “Feedback Form” link sent through email.
2. Click on Star Rating to select a rating.
3. Click on the Comment Box to enter feedback.
4. Type Comments .
5. Click "Submit."

Data Entry – 2 mouse clicks & 50 keystrokes

1. Click on Star Rating to select a rating.
2. Click in the Comment Box to enter feedback.
3. Type the Comments.
4. Click "Submit."

Scan QR (UC-7) – total 1 mouse click

1. Click "Scan QR" on the Home page.

Data Entry – 0 keystrokes

1. Hold QR code up to the scanner.

View History (UC-8) – total 2 mouse clicks

1. Click "Visitor History" on the Home page.
2. Click in the Search field to enter Visitor ID or Name.

Data Entry – 1 mouse click & 12 keystrokes

1. Click in the Search field to enter Visitor ID or Name.
2. Type Visitor ID or Name.

Manage Blacklist (UC-9)

Add To Blacklist (UC-10) – total 4 mouse clicks

1. Click "Manage Blacklist" on the Home page.
2. Click in the Search field to enter Visitor Name.
3. Click the "+" (add) icon to add a record.
4. Click "Confirm."

Remove From Blacklist (UC-11) – total 4 mouse clicks

5. Click "Manage Blacklist" on the Home page.
6. Click in the Search field to enter Visitor Name.
7. Click the "x" (delete) icon to remove a record.
8. Click "Confirm."

Manage User (UC-12)

Add User (UC-13) – total 1 mouse clicks

1. Click “Create User” on the home screen
2. Enter username, password and phone number
3. Select user status
4. Select the role of the user to be added.

Data Entry – 12 mouse clicks & 40 keystrokes

If the selected role is a resident:

1. Click in the First Name field and enter information.
2. Click in the Last Name field and enter information.
3. Click in the Phone Number field and enter information.
4. Click in the House Number field and enter information.
5. Click in the Address field and enter information.
6. Click “Add” to add the user.

If the selected role is a security guard:

1. Click in the First Name field and enter information.
2. Click in the Last Name field and enter information.
3. Click in the Phone Number field and enter information.
4. Click in the Shift field and select shift.
5. Click in the Access Point field and select the access point.
6. Click “Add” to add the user.

Edit User (UC-14) – total 3 mouse clicks

1. Click "List of Users" on the Home page.
2. Click on Search field
3. Click on the Edit icon located on the actions column

Data Entry – 22 mouse clicks & 40 keystrokes

To edit a resident:

1. Click on Username and enter the updated username.
2. Click on Password and enter the updated password.
3. Click on Phone Number and enter the updated phone number.
4. Click on Status and select the status.
5. Click on Role and select the role.

6. Click in the First Name field and enter the updated first name.
7. Click in the Last Name field and enter the updated last name.
8. Click in the Phone Number field and enter the updated phone number.
9. Click in the House Number field and enter the updated house number.
10. Click in the Address field and enter the updated address.
11. Click “Update” to save changes.

To edit a security guard:

1. Click on Username and enter the updated username.
2. Click on Password and enter the updated password.
3. Click on Phone Number and enter the updated phone number.
4. Click on Status and select the updated status.
5. Click on Role and select the updated role.
6. Click in the First Name field and enter the updated first name.
7. Click in the Last Name field and enter the updated last name.
8. Click in the Phone Number field and enter the updated phone number.
9. Click on Shift and select the updated shift.
10. Click on Access Point and select the updated access point.
11. Click “Update” to save changes.

Delete User (UC-15) – total 3 mouse clicks & 15 keystrokes

1. Click "List of Users" on the Home page.
2. Click on the Search field and enter the User ID or name.
3. Click on the Delete icon in the actions column.
4. Click “Confirmation” to confirm deletion.

View Feedback (UC-16)

Navigation – total 2 mouse clicks

1. Click "Visitor Feedback" on the Home page.
2. Click the “View more” button to view more feedback.

Data Entry – 0 keystrokes

1. No keystrokes required for this action

Using Use Case Points

All your actors use a GUI, which makes them complex:

Actor	Type	Weight
Resident	Complex	3
Visitor	Complex	3
Security Guard	Complex	3
Administrator	Complex	3
Total UAW	= 4 actors × 3 = 12	

Table 16. Actor Classification(UAW)

Use Case	Interaction Level	Classification	Weight
UC-2 CreateVisitorSchedule	~13+ fields, 14 clicks, 100+ keystrokes	Complex	15
UC-6 Submit Feedback	~5 clicks, 50 keystrokes	Simple	5
UC-7 Scan QR	1 click, no typing	Simple	5
UC-8 View History	2 clicks, minimal typing	Simple	5
UC-9 Manage Blacklist	Combined with UC-10/11 – ~4 clicks	Simple	5
UC-12 Manage Users	Grouped (UC-13/14/15)	Complex	15
UC-16 View Feedback	2 clicks, no typing	Simple	5
Total UUCW	= 15 + 5 + 5 + 5 + 5 + 15 + 5 = 55		

Table 17. Use Case Classification (UUCW)

Unadjusted Use Case Point(UUCP)

UUCP = 55+12

= 67

TF No.	Technical Factor	Weight	Est. Rating	Score
TF1	Distributed system	2.0	2	4.0
TF2	Response time or performance objectives	1.0	3	3.0
TF3	End-user efficiency	1.0	4	4.0
TF4	Complex internal processing	1.0	3	3.0
TF5	Reusability	1.0	2	2.0
TF6	Easy to install	0.5	3	1.5
TF7	Easy to use	0.5	5	2.5
TF8	Portable	2.0	1	2.0
TF9	Easy to change	1.0	3	3.0
TF10	Concurrent use	1.0	2	2.0
TF11	Includes special security features	1.0	4	4.0
TF12	Provides direct access for third parties	1.0	1	1.0
TF13	Special user training needed	1.0	1	1.0
Total TF		= 33		
TCF		= 0.6 + (0.01 × Total TF) = 0.6 + (0.01 × 33) = 0.6 + 0.33 = 0.93		

Table 18. Technical Complexity Factor (TCF)

EF No.	Environmental Factor	Weight	Est. Rating	Score
EF1	Familiar with the development process	1.5	3	4.5
EF2	Application experience	0.5	2	1.0
EF3	Object-oriented experience	1.0	3	3.0
EF4	Lead analyst capability	0.5	3	1.5
EF5	Motivation	1.0	4	4.0
EF6	Stable requirements	2.0	4	8.0
EF7	Part-time staff	1.0	2	2.0
EF8	Difficult programming language or environment	1.0	1	1.0
Total EF		= 25		
ECF		= 1.4 + (−0.03 × Total EF) = 1.4 + (−0.03 × 25) = 1.4 − 0.57 =0.65		

Table 19. Environmental Factor (EF)

Final Use Case Points (UCP)

= UUCP × TCF × ECF

= 67 × 0.93 × 0.65

= 40.5015

Duration Estimation

Using Productivity Factor (PF) = 28 hours per UCP:

Effort = UCP × PF

= 40.5015 × 28

= 1,134.042 hours

System Architecture

Identifying Subsystems

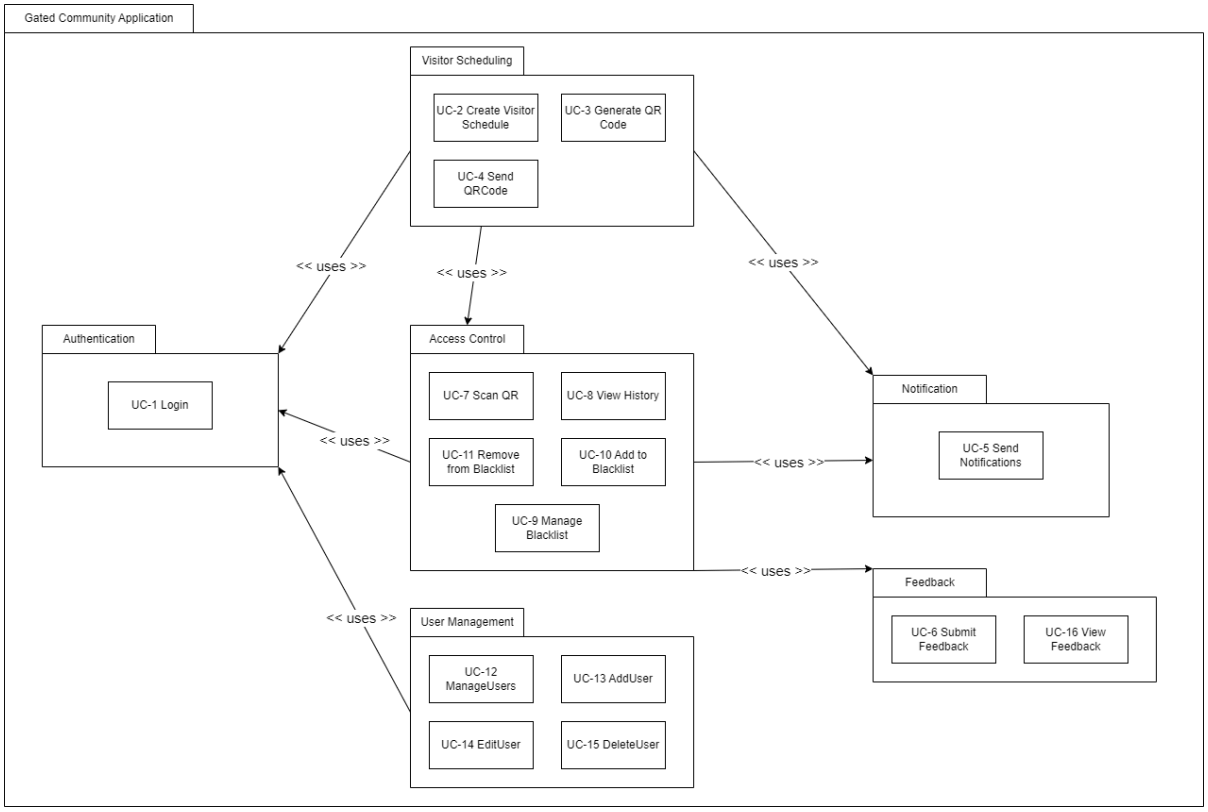

Fig 24. UML Package of Subsystems

The diagram shows the subsystem design for the Gated Community Application. It organizes related use cases into functional groups: Authentication, Visitor Scheduling, Access Control, User Management, Notification, and Feedback. Users must first log in through the Authentication subsystem (UC-1) to access features.

Visitor Scheduling allows residents to create schedules (UC-2), generate QR codes (UC-3), and send them (UC-4), which are later used by Access Control for scanning (UC-7) and managing visitor access (UC-8 to UC-11). Access Control also interacts with Notification (UC-5) to alert users and with Feedback (UC-6, UC-16) to collect and display visitor feedback.

User Management (UC-12 to UC-15) lets administrators handle user accounts. The subsystems are connected using relationships, showing dependency and interaction, ensuring secure and coordinated system behavior.

Architecture Styles

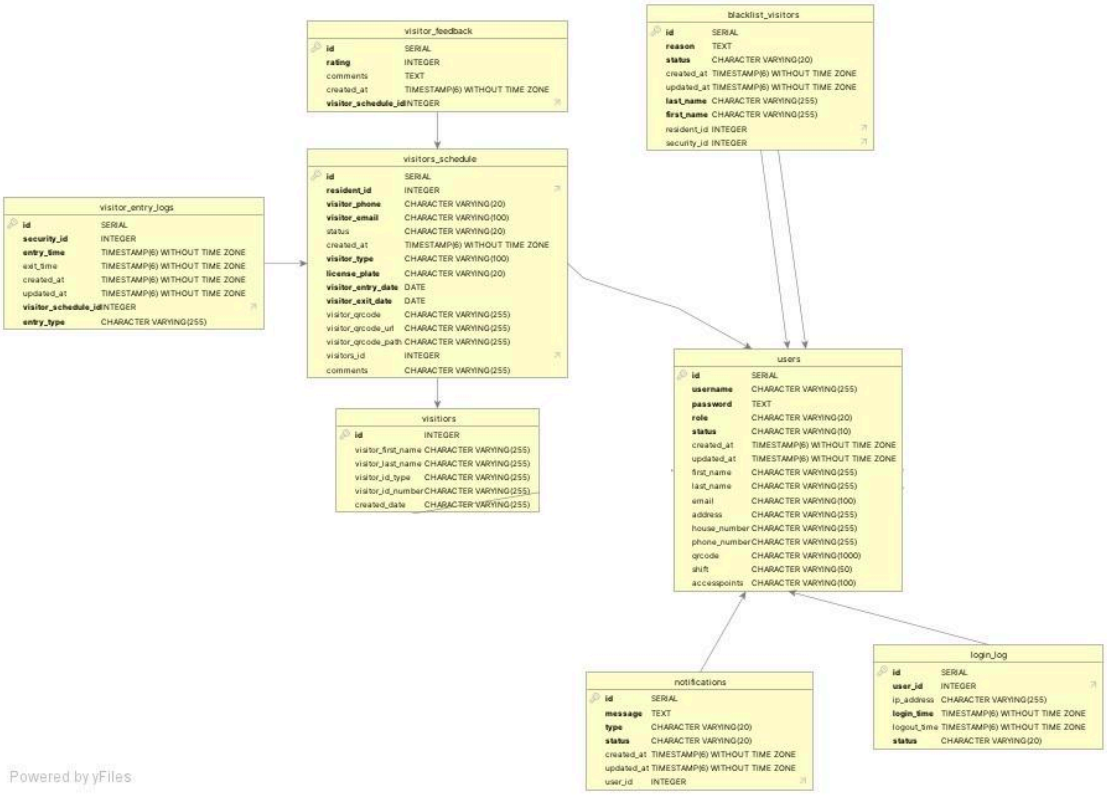

Fig 25. Database Schema

Based on Figure 25, the architectural design of the gated community system primarily follows a layered architecture with elements of a client-server model and event-driven architecture.

1. Data Layer
   The schema defines structured entities for managing users, visitors, vehicles, security logs, notifications, and access control. Tables such as users, security, and residents define user roles, while visitor_schedule, and visitor_entry_logs track visitor movements. The use of relational database design ensures data integrity, with foreign key relationships linking residents, security personnel, and visitors.

2. Business Logic Layer
   The system's logic enforces access control through relationships between security personnel, residents, and access points. Tables such as blacklist_visitors and notifications suggest an event-driven approach, where actions such as unauthorized visitor entries trigger notifications and log updates. The security team can manage access dynamically, ensuring real-time response to events.

3. Presentation Layer
   The web-based interface allows users to interact with the system, submit visitor

schedules, and review logs. The system ensures residents manage their visitor_schedule, and administrators oversee the entire operation.

By combining a relational database model with event-driven mechanisms, the system ensures secure, scalable, and real-time management of visitor access, resident activities, and security enforcement.

Mapping Systems to Hardware

As a web-based, mobile-responsive application, it will allow residents to access features from their smartphones, tablets, or computers once an internet connection is available. Security personnel will use the system at multiple stations if the community has multiple entries and exit points, accessing it via devices like tablets or desktops to verify visitors and log incidents. Administrators can manage the system from their offices, overseeing user accounts, blacklists, and system operations. A centralized web server will handle authentication, data storage, and notifications, ensuring real-time synchronization and secure access across all user roles.

Connectors and Network Protocols

The application will use Prisma alongside Next.js, utilizing built-in API routes for seamless communication between the frontend and backend. Prisma, an ORM (Object-Relational Mapping) tool, will handle database interactions efficiently, connecting to a PostgreSQL database. This choice is ideal for our system because Prisma simplifies complex database queries, improves security by reducing the risk of SQL injection, and enhances maintainability with its type-safe query system. Next.js API routes allow for a structured, scalable approach to handling requests, making it easier to manage authentication, data retrieval, and other backend operations essential for our gated community application.

Global Control Flow

Our gated community application is event-driven, meaning it does not follow a strict linear sequence of steps for every user. Instead, it responds to user actions, such as residents generating QR codes, and security personnel scanning visitor passes. Each user can interact with the system in a different order depending on their needs.

Regarding time dependency, the system includes some timed operations. For example, it ensures that QR codes for recurring visitors expire after their scheduled time, and it locks user accounts for 30 minutes after multiple failed login attempts. Additionally, real-time alerts and notifications are sent instantly to residents and security personnel when a visitor checks in or is denied entry.

The system operates in an event-response model rather than a strict real-time system. However, certain actions require real-time performance, such as QR code scanning, blacklist alerts, and security logs, which must be processed immediately to ensure smooth entry management.

Hardware Requirements

Our system relies on several critical resources to ensure smooth operation and optimal performance. The server must have at least 32GB of memory and a RAID 5 configuration with three 500GB hard drives to support data redundancy, security, and efficient storage management. A stable and high-speed network connection is essential for seamless communication between users and the database. Security personnel will utilize 10-inch tablets for scanning visitor QR codes, which require responsive touchscreens and reliable internet connectivity for real-time verification. Additionally, the system depends on modern web browsers for residents and administrators to access the platform, with a recommended minimum screen resolution of 1280 × 720 pixels for an optimal user experience.

Analysis and Domain Modeling

Concept Model

I. Concept Definitions

Types “D” and “K” denote doing vs. knowing responsibilities, respectively.

Responsibility Description	Type	Concept Name
Allow the resident to input visitor details, including visit frequency and duration.	D	Controller
Store visitor details such as name, frequency, and duration of visits.	K	RecordStorage
Generate a persistent QR code for the visitor.	D	QRCodeManager
Store generated QR codes for visitor identification.	K	RecordStorage
Notify the security team about the new recurring visitor.	D	NotificationController
Store visit records in the database for future reference.	D	VisitManager
Display creation form to user.	D	UIManager

Table 20. Concept Definition for CreateVisitSchedule (UC-2)

Responsibility Description	Type	Concept Name
Coordinate the submission and storage of feedback responses.	D	Controller
Store visitor check-in data and associated feedback.	K	RecordStorage
Process and log visitor feedback.	D	FeedbackManager
Display confirmation message upon successful feedback submission.	D	UIManager

Table 21. Concept Definition for SubmitFeedback (UC-6)

Responsibility Description	Type	Concept Name
Coordinate the QR scanning and validation process.	D	QRCodeManager
Store registered visitor QR codes and check-in data.	K	RecordStorage
Validate scanned QR codes against registered visitors.	D	ValidationManager
Log all visitor check-in attempts.	D	ActionLogger

Table 22. Concept Definition for ScanQR(UC-7)

Responsibility Description	Type	Concept Name
Coordinate actions for searching and displaying past visitor entries.	D	DataRetrievalManager
Store visitor check-in data for retrieval and display.	K	RecordStorage
Retrieve past visitor records based on search criteria (filters).	D	DataRetrievalManager
Display the search interface for entering filter criteria.	D	UIManager
Validate search inputs to ensure correct filtering of data.	D	ValidationManager
Process the retrieval and display of visitor history based on the security guard’s inputs.	D	DataRetrievalManager
Notify the user if no records are found or if there is a system issue retrieving the data.	D	NotificationController

Table 23. Concept Definition forViewHistory (UC-8)

Responsibility Description	Type	Concept Name
Coordinate actions related to adding or removing a visitor from the blacklist.	D	BlacklistManager
Store visitor data and their blacklist status in the system.	K	RecordStorage
Retrieve a visitor's profile when searching by name or ID.	D	DataRetrievalManager
Display the list of visitors and search options.	D	UIManager
Validate whether the visitor exists in the system when performing blacklist actions.	D	ValidationManager
Add the visitor to the blacklist or remove them from it based on the user's selection.	D	BlacklistManager
Notify the user if no visitor records are found or if there is a system issue retrieving data.	D	NotificationController

Table 24. Concept Definition for ManageBlacklist(UC-9)

Responsibility Description	Type	Concept Name
Coordinate actions for managing user accounts (add, edit, delete).	D	UserManagement Controller
Store user account details in the system.	K	RecordStorage
Retrieve user account details for editing or deleting.	D	DataRetrievalManager
Display the relevant interface for adding, editing, or deleting a user.	D	UIManager
Process the request (add, edit, or delete a user) based on the administrator's input.	D	UserManagement Controller
Display a confirmation message after a successful action (add, edit, or delete).	D	UIManager
Handle missing required fields during user creation or modification and show an error message.	D	UIManager

Table 25. Concept Definition for ManageUsers (UC-12)

Responsibility Description	Type	Concept Name
Handle the process of retrieving and showing visitor feedback.	D	FeedbackManager
Container for storing and retrieving feedback data submitted by visitors.	K	RecordStorage
Query the database to fetch the feedback entries for viewing.	D	DataRetrievalManager
Display the retrieved feedback to the user.	D	UIManager
Store the status of whether feedback exists in the system or not (for notifications).	K	RecordStorage
Notify the user when no feedback is available.	D	NotificationController
Provide the option to retry fetching feedback after an unsuccessful attempt.	D	UIManager
Allow the user to exit the feedback viewing section.	D	UIManager

Table 26. Concept Definition for ViewFeedback (UC-16)

II. Association Definitions

Concept Pair	Association Description	Association Name
UI Manager ↔ Controller	UIManager retrieves the information from the client to be processed within the system.	Retrieves data
Controller ↔ RecordStorage	Controller stores visitor details such as name, frequency, and duration in the RecordStorage.	Stores visitor details
Controller ↔ QRCodeManager	Controller generates a persistent QR code for the visitor.	Generates QR Code
QRCodeManager ↔ RecordStorage	QRCodeManager stores the generated QR codes for visitor identification.	Stores QR Code
Controller ↔ NotificationController	Controller notifies the security team about the new recurring visitor.	Sends notification
Controller ↔ VisitManager	Controller stores visit records in the database for future reference.	Stores visit records
VisitManager ↔ RecordStorage	VisitManager stores visit records for historical reference.	Stores visit history

Table 27. Association Definition for CreateVisitSchedule (UC-2)

Concept Pair	Association Description	Association Name
FeedbackManager ↔ RecordStorage	Controller stores feedback responses and visitor check-in data in RecordStorage.	Stores feedback data
Controller ↔ FeedbackManager	Controller processes visitor feedback.	Process feedback
Controller ↔ UIManager	Controller displays a confirmation message upon successful feedback submission.	Displays confirmation

Table 28. Association Definition for SubmitFeedback (UC-6)

Concept Pair	Association Description	Association Name
QRCodeManager ↔ RecordStorage	QRCodeManager stores registered visitor QR codes and check-in data.	Stores QR data
QRCodeManager ↔ ValidationManager	QRCodeManager validates scanned QR codes against registered visitors.	Validates QR code
QRCodeManager ↔ ActionLogger	QRCodeManager logs all visitor check-in attempts.	Logs check-in

Table 29. Association Definition for ScanQR(UC-7)

Concept Pair	Association Description	Association Name
DataRetrievalManager ↔ RecordStorage	DataRetrievalManager retrieves visitor check-in data for display.	Retrieves check-in data
DataRetrievalManager ↔ UIManager	DataRetrievalManager displays the search interface for entering filter criteria.	Displays search interface
DataRetrievalManager ↔ ValidationManager	DataRetrievalManager validates search inputs to ensure correct filtering of data.	Validates search input
ValidationManager↔ NotificationController	ValidationManger notifies the user if no records are found or if there is a system issue.	Sends notification

Table 30. Association Definition for ViewHistory (UC-8)

Concept Pair	Association Description	Association Name
BlacklistManager ↔ RecordStorage	BlacklistManager stores visitor data and their blacklist status.	Stores blacklist status
DataRetrievalManager ↔ BlacklistManager	DataRetrievalManager retrieves a visitor's profile when searching by name or ID.	Retrieves visitor profile
BlacklistManager ↔ UIManager	BlacklistManager displays the list of visitors and search options.	Displays visitor list
BlacklistManager ↔ ValidationManager	BlacklistManager validates whether the visitor exists in the system.	Displays visitor list
ValidationManager ↔ NotificationController	ValidationManager notifies the user if no visitor records are found or if there is a retrieval issue.	Sends notification

Table 31. Association Definition for ManageBlacklist(UC-9)

Concept Pair	Association Description	Association Name
UserManagementController ↔ RecordStorage	UserManagementController stores user account details.	Stores user details
UserManagementController ↔ DataRetrievalManager	UserManagementController retrieves user account details for editing or deleting.	Retrieves user details
UserManagementController ↔ UIManager	UserManagementController displays the interface for adding, editing, or deleting a user.	Displays user interface

Table 32. Association Definition for ManageUsers (UC-12)

Concept Pair	Association Description	Association Name
FeedbackManager ↔ RecordStorage	FeedbackManager stores and retrieves feedback data submitted by visitors.	Stores feedback
FeedbackManager ↔ DataRetrievalManager	FeedbackManager queries the database to fetch feedback entries.	Retrieves feedback
FeedbackManager ↔ UIManager	FeedbackManager displays the retrieved feedback to the user.	Displays feedback
UIManager↔ NotificationController	FeedbackManager notifies the user if no feedback is available.	Sends feedback notification

Table 33. Association Definition for ViewFeedback (UC-16)

III. Attribute Definitions

Concept	Attributes	Attribute Description
ActionLogger	LogID, UserID, ActionType, Timestamp	Tracks system actions and records them for auditing.
BlacklistManager	BlacklistID, VisitorID, UserID, Status, Reason, CreatedAt, UpdatedAt	Handles blacklisting and un-blacklisting of visitors.
Controller		Generalized controller coordinating use case actions.
DataRetrievalManager	QueryID, SearchCriteria, ResultData	Handles searching and retrieving stored data.
FeedbackManager	FeedbackID, VisitorManagerID, Rating, CommentsCreatedAt	Manages feedback submission, retrieval, and notifications.
NotificationController	NotificationID, UserID, Message, Type, Status, CreatedAt, UpdatedAt	Sends alerts and notifications to users.
QRCodeManager	QRCodeID, VisitorID,	Manages the generation and validation

	QRCode, QRCodeURL, QRCodePath, ExpirationDate	of QR codes.
RecordStorage	RecordID, Data, Timestamp	Stores records such as visitor logs, feedback, and system actions.
UIManager		Manages user interfaces and displays relevant information.
UserManagementController	UserID, Username, Password, Role, Status, FirstName, LastName, Email, PhoneNumber	Manages user accounts and permissions.
ValidationManager		Ensures correctness of user input and system data.
VisitManager	VisitID, UserID, VisitorID, EntryDate, ExitDate, Status	Handles visitor check-ins, scheduling, and tracking.

Table 34. General Attribute Definitions

IV. Traceability Matrix

Use Case	PW	Action Logger	Blacklist Manager	Controller	Data Retrieval Manager	Feedback Manager	Notification Controller	QRCode Manager	Record Storage	UI Manager	User Management Controller	Validation Manager	Visit Manager
UC-2	32			X			X	X	X				X
UC-6	19			X		X	X		X	X			
UC-7	35	X						X	X			X	
UC-8	16				X		X		X	X		X	
UC-9	22		X		X		X		X	X		X	
UC -12	28				X				X	X	X	X	
UC -16	16				X	X	X		X	X			

Table 35. Table Showing Domain Analysis Traceability Matrix

V. Domain Model

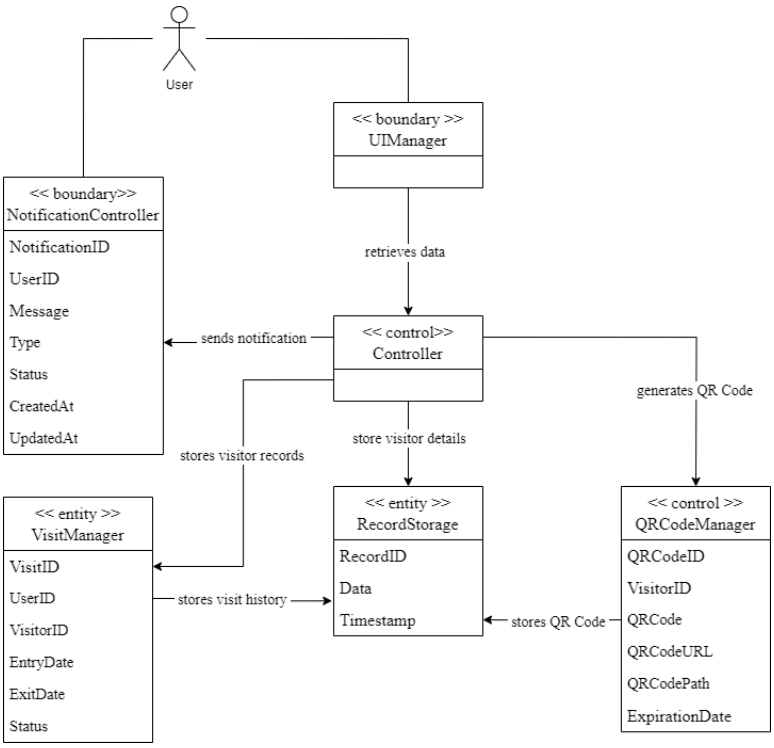

Figure 26. Domain Model for CreateVisitSchedule (UC-2)

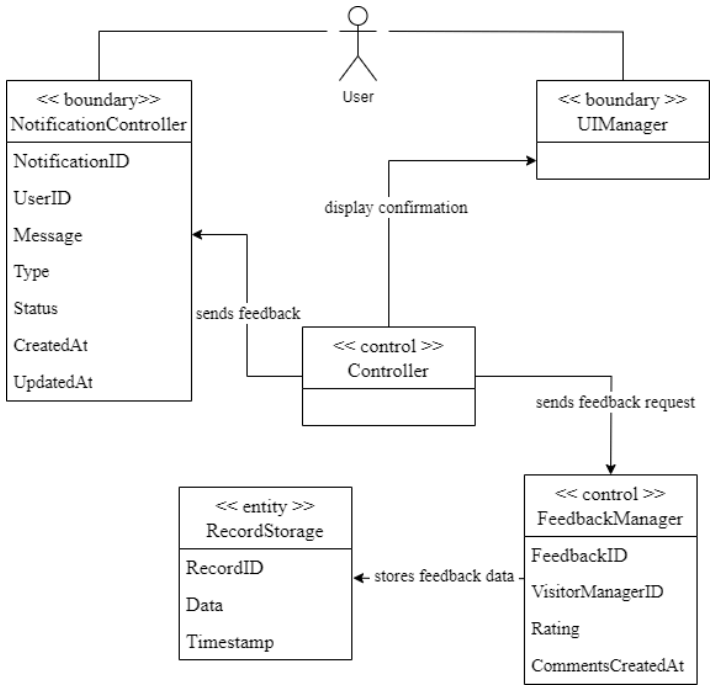

Figure 27. Domain Model for SubmitFeedback (UC-6)

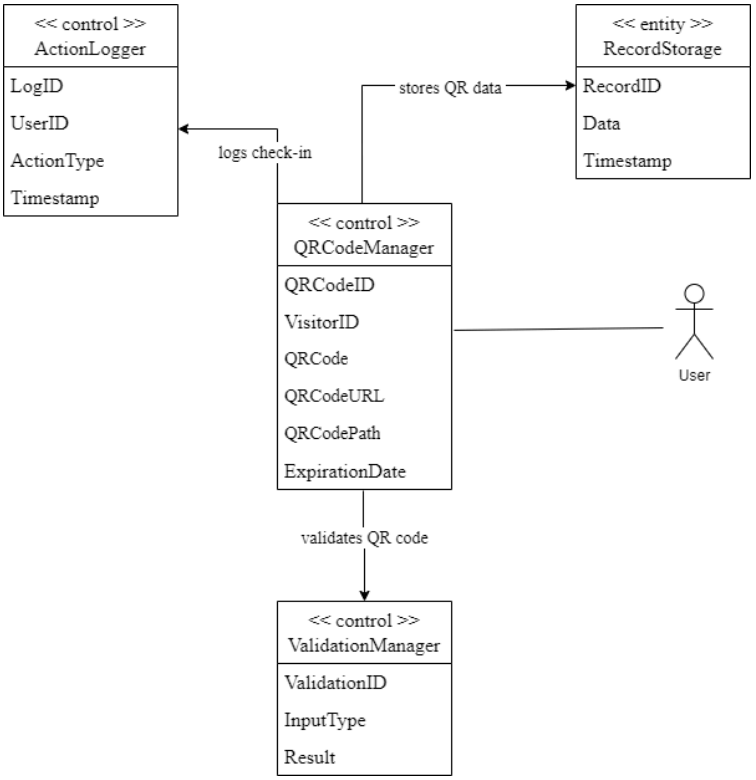

Figure 28. Domain Model for ScanQR(UC-7)

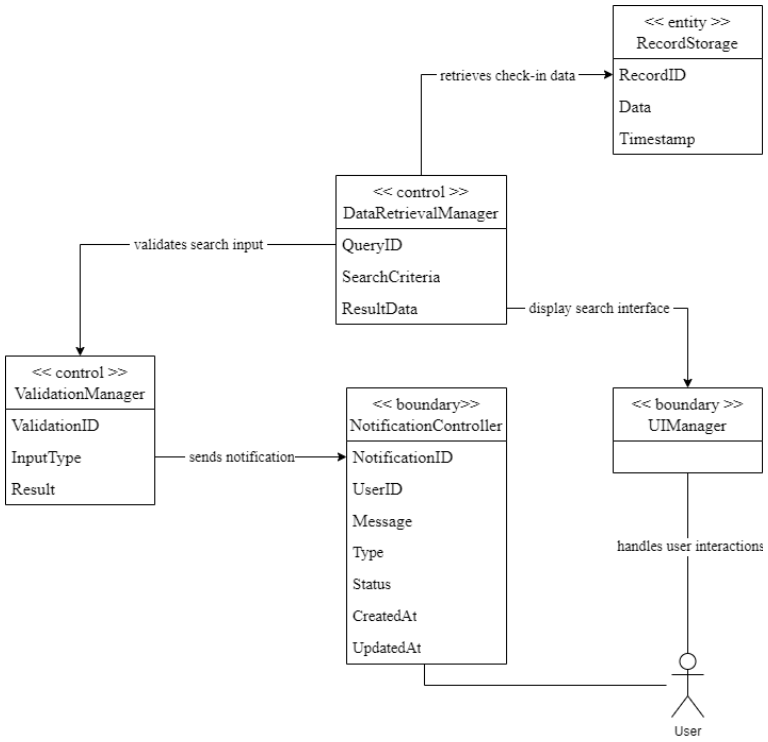

Figure 29. Domain Model forViewHistory (UC-8)

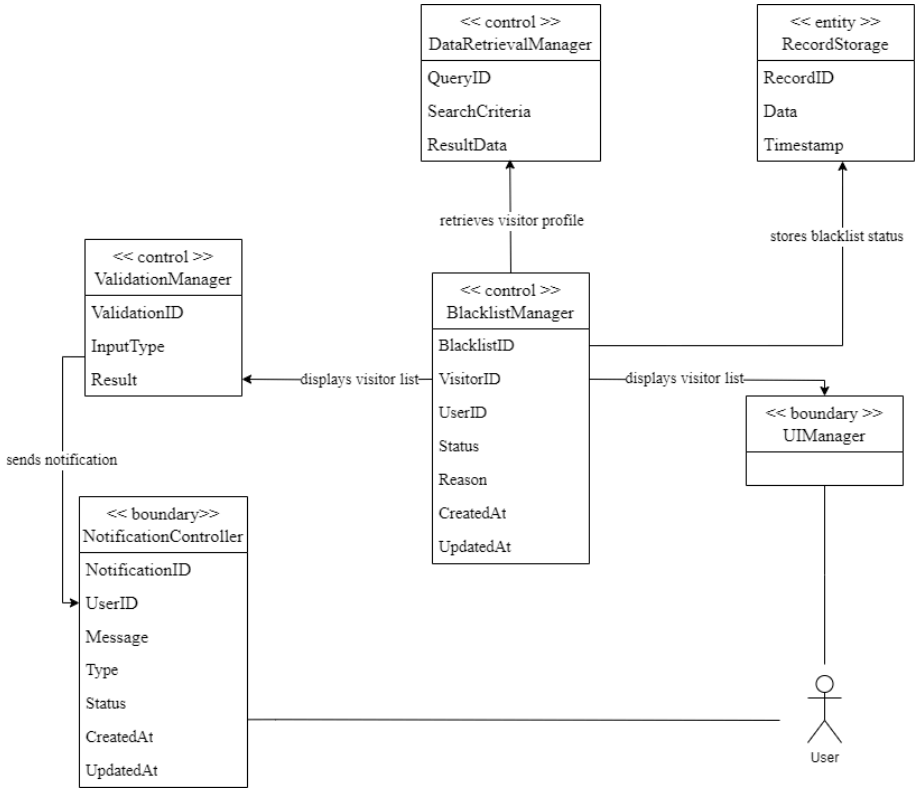

Figure 30. Domain Model for ManageBlacklist(UC-9)

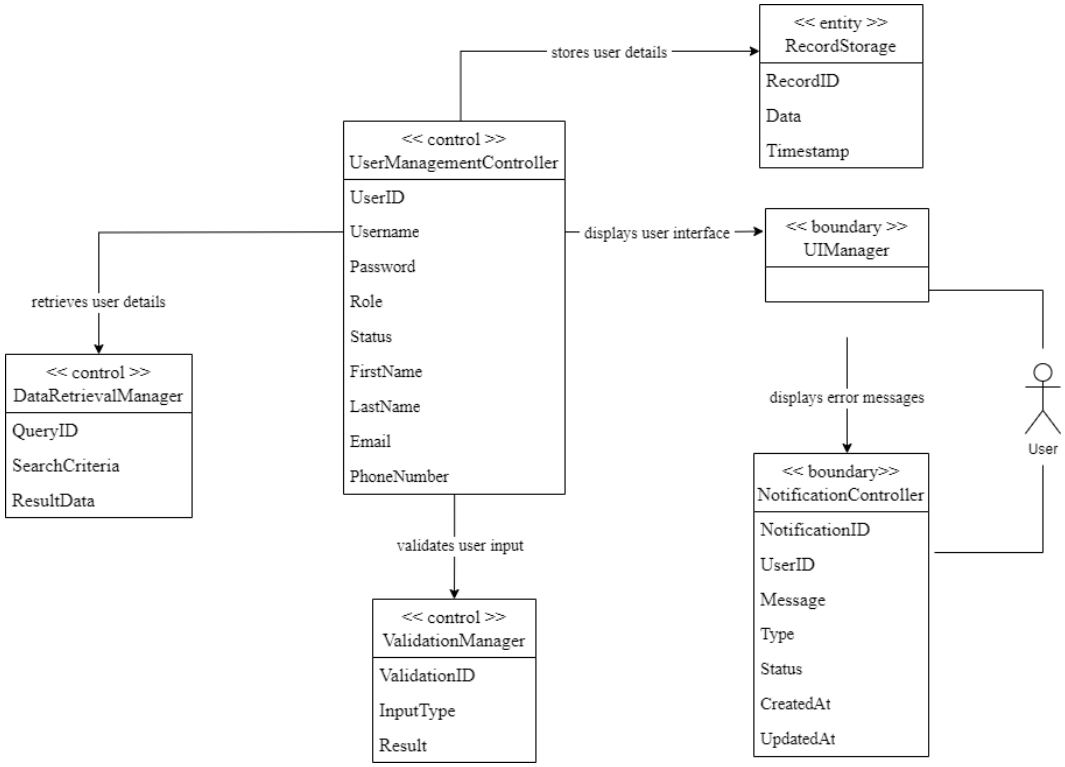

Figure 31. Domain Model for ManageUsers (UC-12)

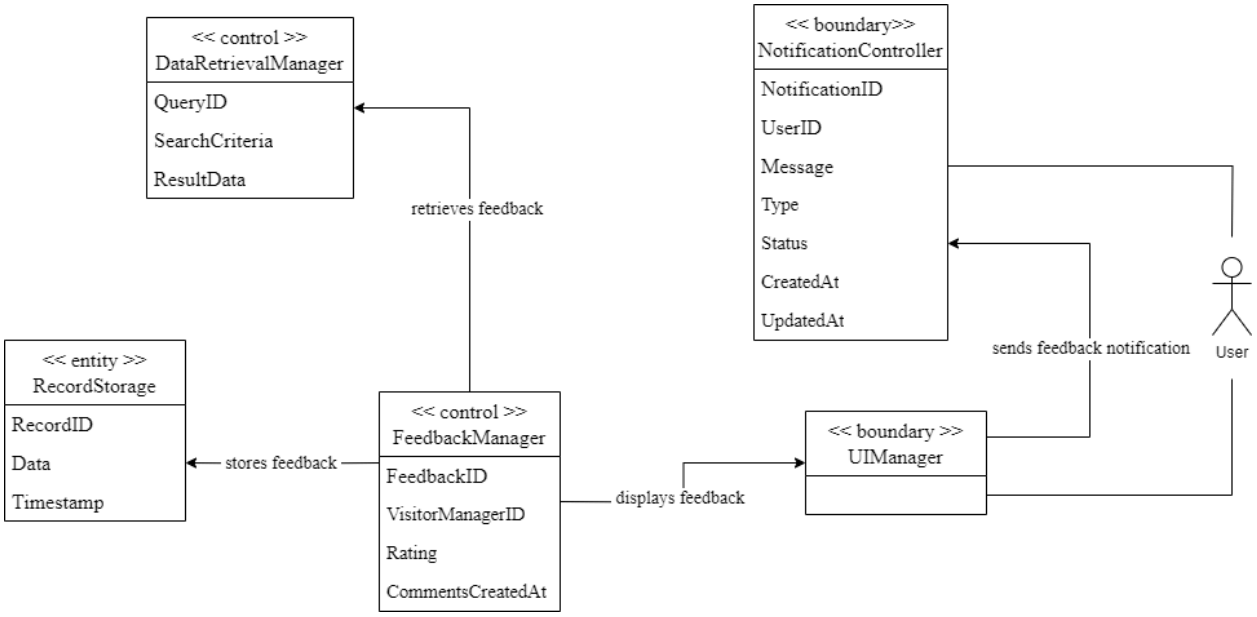

Figure 32. Domain Model for ViewFeedback (UC-16)

System Operation Contracts

Contract Name	CreateVisitSchedule
Operation	registerVisitor(visitorDetails, schedule)
Actor:	Resident
Cross Reference	UC-2 (CreateVisitorSchedule)
Responsibilities	● Validate visit frequency/duration against community rules (REQ-20). ● Encode QR code with visitor ID and access timeframe (NONREQ-13). ● Automate email/SMS delivery of QR code to visitor (UC-4).
Preconditions	● Resident is authenticated (NONREQ-8 enforced). ● Visitor details (name, contact, frequency, duration) comply with input validation rules (REQ-16, REQ-17).
Post Conditions	● A new Visitor object is created in the database with attributes including visitor name, contact information, and access permissions (REQ-22) ● A Schedule object linked to the Visitor is created with frequency and duration attributes ● A unique QR code object is generated containing encrypted visitor ID and temporal access parameters (UC-3) ● A Notification object is created and sent to the Security team with visitor details (REQ-21) ● The Resident's visitorList collection is updated to include the new Visitor object

Table 36. System Contracts for CreateVisitSchedule (UC-2)

Contract Name	SubmitFeedback
Operation	submitFeedback(visitorID, rating, comments)
Actor	Visitor
Cross Reference	UC-6 (SubmitFeedback)
Responsibilities	● Enforce feedback form validation (e.g., rating 1–5, max 500 characters). ● Anonymize feedback for compliance with NONREQ-16.
Preconditions	● Visitor check-in data exists (REQ-3). ● Feedback link accessed via valid session (NONREQ-15).
Post Conditions	● A new Feedback object is created with attributes: visitorID, rating (1-5),

	comments (≤500 chars), timestamp, and anonymized flag ● The Visitor object is updated with a hasFeedback attribute set to true ● A FeedbackConfirmation object is created and displayed to the visitor (NONREQ-17) ● The system's feedbackCollection is incremented and updated with the new Feedback object (REQ-23)

Table 37. System Contracts for SubmitFeedback (UC-6)

Contract Name	ScanQR
Operation	scanQR(qrCode)
Actor	Security Guard
Cross Reference	UC-7 (ScanQR)
Responsibilities	● Decrypt QR code and match with registered visitors (REQ-13). ● Verify access validity (e.g., recurring vs. one-time visits). ● Trigger real-time notifications to residents (NONREQ-9).
Preconditions	● QR code format is valid (REQ-12). ● Security guard has system access (NONREQ-6).
Post Conditions	● A new VisitorLog object is created with attributes: visitorID, timestamp, entryPoint, and securityGuardID (REQ-14) ● The Visitor object's lastCheckIn attribute is updated with current timestamp ● An AccessRecord object is created with attributes: accessGranted (boolean), timestamp, and reason (if denied) (NONREQ-14) ● The QR code object's status is updated to "used" if it's a one-time code

Table 38. System Contracts for ScanQR(UC-7)

Contract Name	ViewHistory
Operation	viewVisitorHistory(filters)
Actor	Security Guard
Cross Reference	UC-8 (ViewHistory)
Responsibilities	Process filter criteria (date ranges, visitor names) from input validation rules (NONREQ-3) Retrieve records matching: ● Visitor ID/name ● Check-in timestamps ● Access status (granted/denied) Handle empty results by displaying: ● "No records found" for valid queries "Database connection error" for system failures
Preconditions	● Security guard is authenticated (NONREQ-8 enforced) ● Visitor check-in data exists in the database (REQ-11)
Post Conditions	● A HistoryResults collection object is created containing all VisitorLog objects matching the filter criteria ● An AuditRecord object is created with securityGuardID, timestamp, and search parameters used (NONREQ-14) ● The UI view object is updated to display the HistoryResults collection ● The system's queryLog collection is updated with the new search parameters

Table 39. System Contracts for ViewHistory (UC-8)

Contract Name	ManageBlacklist
Operation	fetchBlacklist(visitorID, action)
Actor	Resident/Security Guard
Cross Reference	UC-9 (ManageBlacklist)
Responsibilities	● Restrict blacklist modifications to authorized roles. ● Propagate blacklist updates to check-in validation workflows.
Preconditions	● Requester has blacklist management privileges (REQ-15). ● Visitor exists in the database (REQ-4).
Post Conditions	● The Visitor object's accessStatus attribute is updated to "blacklisted" or "approved" ● A BlacklistRecord object is created with attributes: visitorID, requestorID, timestamp, and action taken ● An AuditLog object is created documenting the blacklist modification (NONREQ-19) ● The system's blacklist collection is updated to include or remove the visitor ● All active QR codes associated with the blacklisted visitor have their validStatus attribute set to false

Table 40. System Contracts for ManageBlacklist(UC-9)

Contract Name	ManageUsers
Operation	modifyUser(userID, action, details)
Actor	Administrator
Cross Reference	UC-12 (ManageUsers)
Responsibilities	● Encrypt sensitive data (e.g., passwords) during storage (NONREQ-19). ● Synchronize user roles with permission matrices.

Preconditions	• Administrator has elevated permissions (REQ-4). • Required fields (e.g., email, role) provided (NONREQ-18).
Post Conditions	• A User object is created/modified/deleted with attributes: userID, encryptedPassword, email, role, and permissions (REQ-15) • The system's userDirectory is updated to reflect the new User object state • A PermissionMatrix object is updated to sync with the User's role attribute • An AdminLog object is created with administrator details and action performed • All subsystems' accessControl objects are updated to reflect the permission changes

Table 41. System Contracts for ManageUsers (UC-12)

Contract Name	ViewFeedback
Operation	fetchFeedback(filters)
Actor	Admin/Resident/Security
Cross Reference	Use cases: ViewFeedback
Responsibilities	• Apply role-based data filtering (e.g., residents see only their visitors). • Handle empty result sets with user-friendly prompts.
Preconditions	• Users have feedback-viewing permissions (REQ-3). • Filters (date, visitor type) are valid (NONREQ-17).
Post Conditions	• A FeedbackResults collection object is created containing all Feedback objects matching the filter criteria • The UI view object is updated to display the filtered FeedbackResults • A ViewLog object is created recording which feedback was accessed and by whom • The system's feedbackViews counter is incremented for analytics • Each displayed Feedback object's viewCount attribute is incremented

Table 42. System Contracts for ViewFeedback (UC-16)

Data Model & Persistent Data Storage

Since we are developing a system that records visitor entries and security logs, it requires data persistence beyond a single execution. The system must store critical information, including users, visitors, schedules, entry logs, and notifications. We will use a relational database to store these key components, ensuring data integrity and security.

The following are the key data components and their storage details:

1. Users: Stores user account information for system access, including security personnel, administrators, and residents.

Attributes: id (Primary Key) – Unique identifier for each user, username – User's login name, password – Securely stored password, role – Defines the user type (e.g., Security, Resident, Admin), status – Account status (active/inactive), first_name, last_name – User's full name, email, phone_number – Contact details, qrcode – Unique QR code assigned to each user, created_at, updated_at – Timestamps for tracking modifications

2. Visitors: Stores details of external visitors to track entry and exit logs.

Attributes:id (Primary Key) – Unique visitor ID, visitor_first_name, visitor_last_name – Visitor's full name, visitor_id_type, visitor_id_number – Identification type and number (e.g., Passport, National ID), created_date – Record creation date

3. Visitor Schedule: Manages visitor entry requests and approvals.

Attributes:id (Primary Key) – Unique schedule ID, resident_id – ID of the resident hosting the visitor (Foreign Key → users.id), visitor_email, visitor_phone – Contact information, visitor_entry_date, visitor_exit_date – Scheduled visit timeframe, license_plate – Vehicle details (if applicable), visitor_qrcode – QR code generated for entry, visitor_qrcode_url, visitor_qrcode_path – Digital QR storage details, status – Approval status (e.g., Pending, Approved, Rejected), created_at – Timestamp for tracking requests

4. Visitor Entry Logs: Tracks real-time visitor check-ins and check-outs.

Attributes:id (Primary Key) – Unique log entry ID, security_id – ID of the security personnel processing the entry (Foreign Key → users.id), visitor_schedule_id – Associated visitor request

(Foreign Key → visitors_schedule.id), entry_time, exit_time – Timestamps for entry and exit, entry_type – Type of entry (e.g., Resident Guest, Delivery)

5. Blacklist Visitors: Stores records of banned visitors.

Attributes:id (Primary Key) – Unique blacklist ID, first_name, last_name – Name of the banned individual, reason – Justification for blacklisting, status – Status of the blacklist (e.g., Active, Expired), resident_id – Associated resident (Foreign Key → visitors.id), security_id – Security personnel who initiated the blacklist (Foreign Key → users.id), created_at, updated_at – Timestamps for record tracking

6. Login Logs: Monitors user access to the system.

Attributes:id (Primary Key) – Unique login attempt ID, user_id – Associated user (Foreign Key → users.id), ip_address – Device IP address used for login, login_time, logout_time – Session timestamps, status – Success or failure of login attempt

7. Notifications: Stores alerts sent to users for approvals, status updates, and reminders.

Attributes:id (Primary Key) – Unique notification ID, user_id – Recipient of the notification (Foreign Key → users.id), message – Notification content, type – Notification category (e.g., Security Alert, Entry Approval), created_at, updated_at – Timestamps for tracking messages

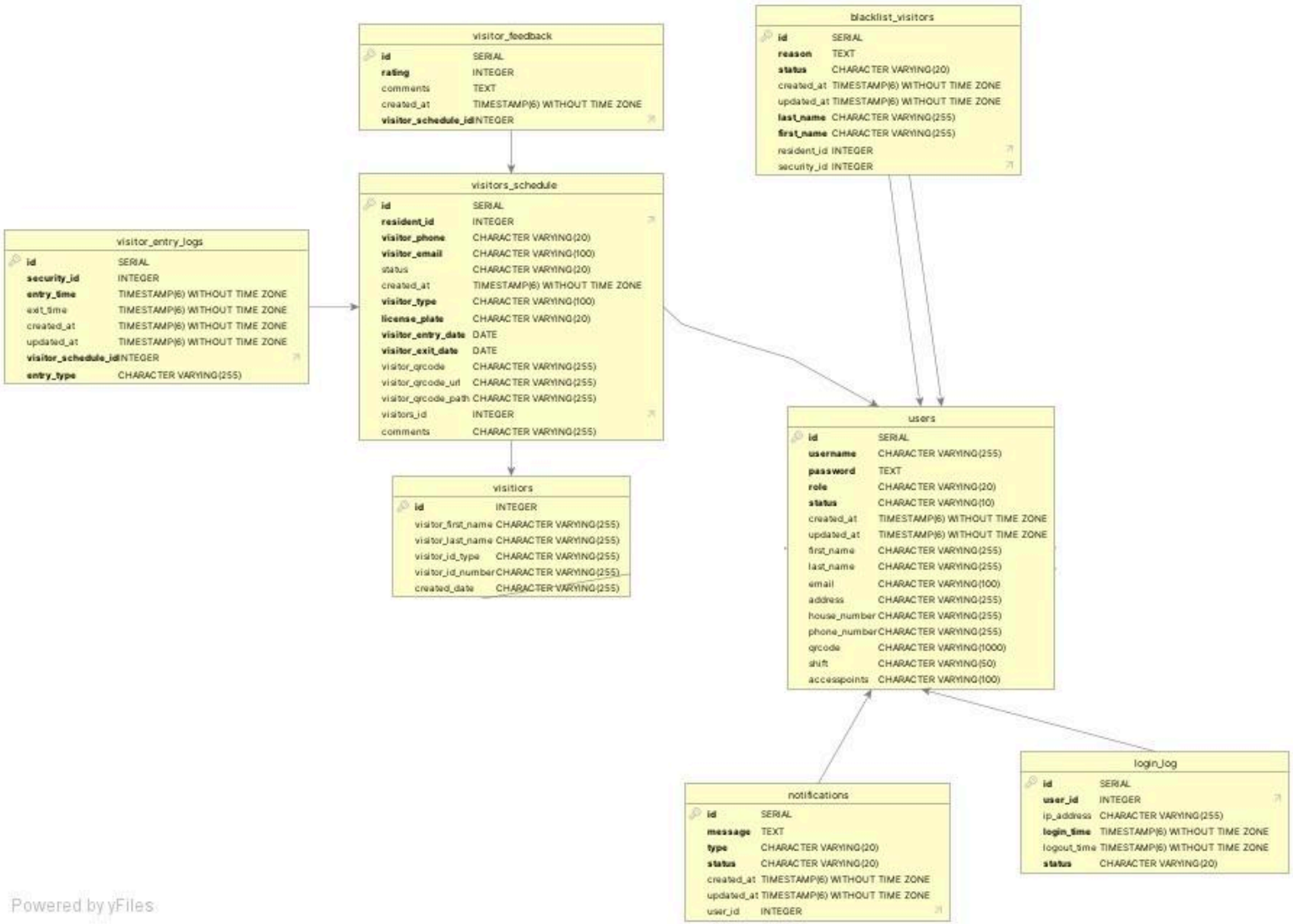

Figure 33. Database Schema

Mathematical Model

Generate a QR Code

- When a resident submits visitor information, the system retrieves the set of input values provided.
- The system then performs a validation check to ensure all required fields are complete and correctly formatted.
- If the information is valid, the system proceeds to encode the visitor details using a structured data format.
- Once encoded, the system generates a QR code containing the visitor data, which can later be scanned at the gate for access authorization.

Verify a QR Code

- The system extracts visitor data from the QR Code.
- The system verifies the visitor's identity, ensures they are on the approved list, and checks their location if applicable.
- Once all verifications pass, the QR Code is accepted, and the visitor's entry is recorded.

Update the scanner interface

- The scanner retrieves the server time.
- Using this time, the system iterates through expected visitor logs to update the interface with details of the current visitor and the next expected visitor.
- The updated interface provides real-time visitor tracking for security personnel.

Check system updates regarding visitor logs and security notifications

- A timer continuously checks the system every minute to update visitor status, track check-ins and check-outs, and send alerts for unauthorized access.
- Any changes in visitor permissions or security alerts trigger immediate updates to the security dashboard.

Interaction Diagrams

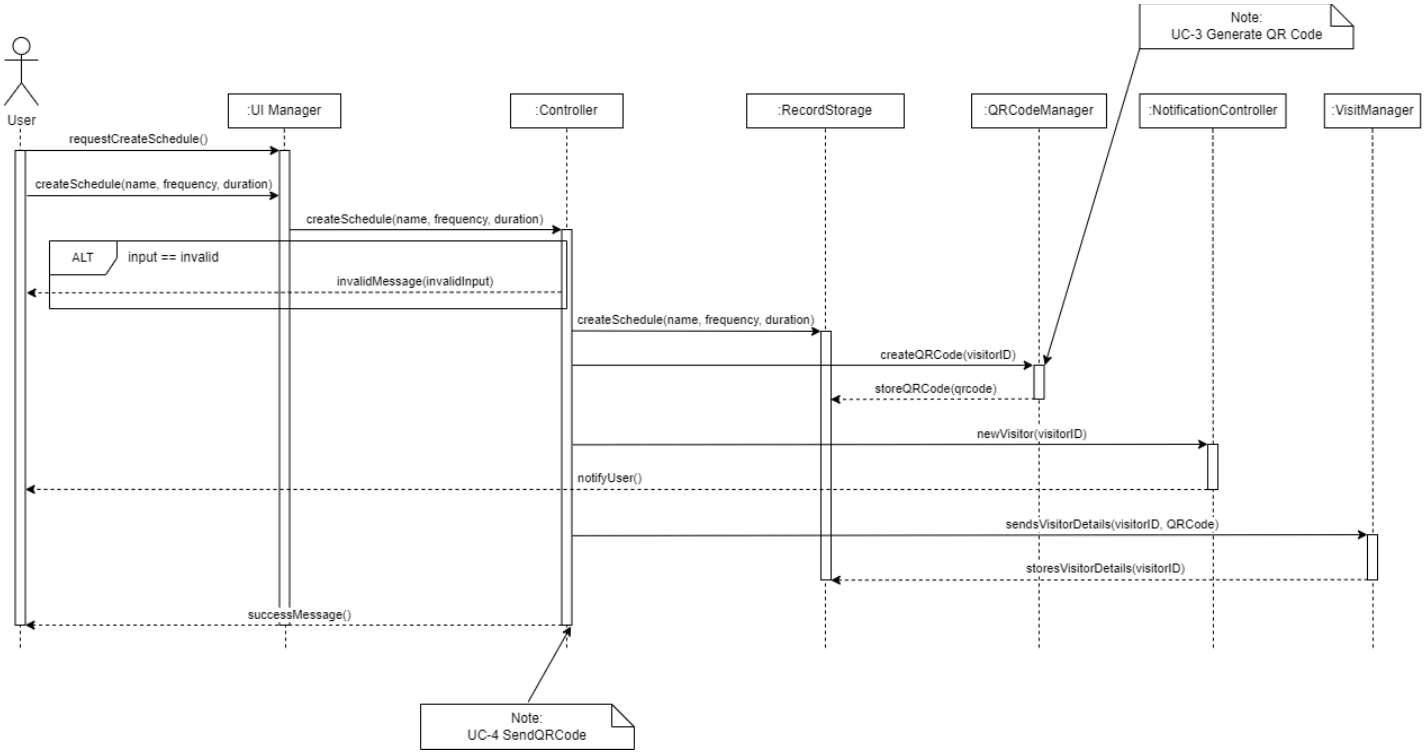

Figure 34. CreateVisitSchedule (UC-2)

When a user wants to create a visit schedule in the gated community system, they begin by requesting the creation of a schedule through the **UIManager**. The **Controller** will then receive the details for processing. The **Controller** first checks whether the provided input is valid. If the input is incorrect or missing required fields, the system triggers an alternate flow, sending an invalid input message back to the **UI Manager**, which then notifies the user about the issue. If the input is valid, the **Controller** proceeds by sending the visit details to **RecordStorage**, where the visit schedule is stored.

Once the visit is successfully recorded, the system moves forward with generating a QR code for the visitor. **RecordStorage** calls the **QRCodeManager**, instructing it to create a QR code for the visitor's unique ID. After generating the QR code, the **QRCodeManager** sends it back to **RecordStorage**, which securely stores the generated code. To ensure that the visit is properly registered, **RecordStorage** then notifies the **VisitManager** about the new visitor. The **VisitManager** forwards the visitor's details, along with the QR code, to the **NotificationController**, which is responsible for notifying the users of the visitor's future arrival.

Once all the necessary details are stored, the system finalizes the process by sending a notification back to the **Controller**, confirming that the visit has been successfully created. The **Controller** then relays a success message to the **UI Manager**, which informs the user that the visit has been scheduled, and the QR code has been generated.

The interaction diagram demonstrates the use of several software design patterns. The Facade Pattern is implemented by the **UI Manager**, which provides a simplified interface to the complex subsystem involving schedule creation and QR code generation. The Controller Pattern is utilized in the **Controller** concept, which handles user requests and manages actions across the system components. The Observer Pattern is represented by the **NotificationController**, which reacts to changes in the system state, such as the successful creation of a schedule, by notifying the user. Lastly, the Singleton Pattern is used in components like **QRCodeManager** and **RecordStorage** to ensure consistent access to shared resources across the application.

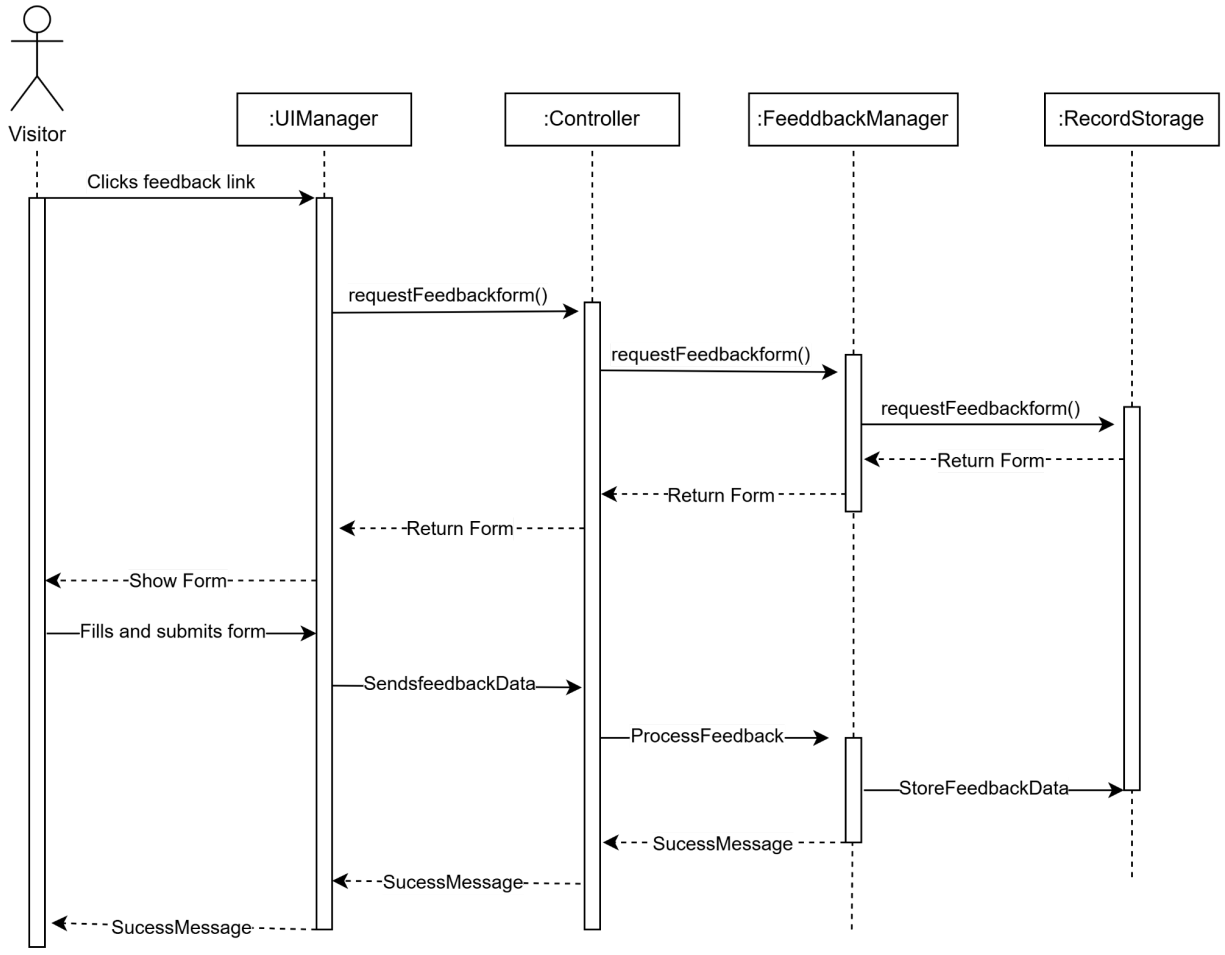

Figure 35. SubmitFeedback (UC-6)

The visitor clicks on the feedback link sent through email. This action prompts the **UIManager** to request the feedback form from the **Controller** using the requestFeedbackForm() function. The **Controller**, in turn, forwards this request to the **FeedbackManager**, which then interacts with **RecordStorage** to retrieve the form. Once the form is retrieved, it is sent back through the same chain, first to the **FeedbackManager**, then to the **Controller**, and finally to the **UIManager**, which displays it to the visitor.

The visitor then fills in the feedback form and submits it. The **UIManager** collects the entered data and sends it to the **Controller** using the sendFeedbackData() function. The **Controller** then forwards this data to the **FeedbackManager**, which processes it using processFeedback(). As part of the processing, the **FeedbackManager** stores the feedback in the system by invoking storeFeedbackData() on **RecordStorage**. Once the data is successfully stored, **RecordStorage** confirms the operation, and a success message is sent back through the same chain, from the **FeedbackManager** to the **Controller**, then to the **UIManager**, which displays the confirmation message to the visitor.

The feedback interaction diagram incorporates several software design patterns. The Facade Pattern is evident in the **UIManager**, which acts as a unified interface to the underlying processes of feedback handling, abstracting complexity from the user. The Controller Pattern is clearly implemented in the **Controller** component, which manages the coordination between the UI, business logic, and data storage. Additionally, the Singleton Pattern is used in both the **FeedbackManager** and **RecordStorage** components to maintain consistent state and centralized access to shared feedback-related resources.

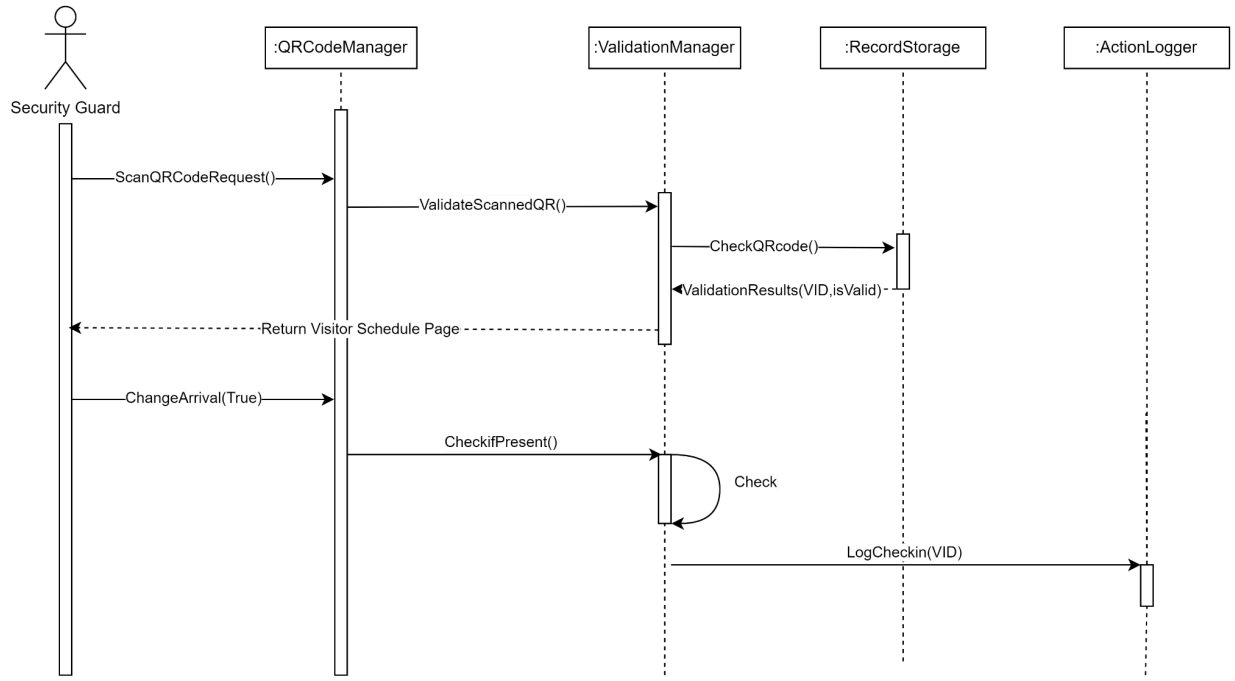

Figure 36.ScanQR(UC-7)

In this process, the **Security Guard** initiates the visitor check-in by triggering the ScanQRCodeRequest() through the **QRCodeManager**. The **QRCodeManager** delegates the request to the **ValidationManager** by calling ValidateScannedQR(), which then interacts with **RecordStorage** using CheckQRCode() to verify the authenticity of the scanned code. Once validation is complete, the **ValidationManager** returns a ValidationResults object containing the Visitor ID (VID) and a flag indicating whether the QR code is valid. Based on the validation, the **QRCodeManager** responds to the **Security Guard** with the Visitor Schedule Page. Upon confirming the visitor's arrival, the **Security Guard** invokes the ChangeArrival(True) method,

prompting the **QRCodeManager** to check if the visitor is in the community via CheckIfPresent(). If the visitor is not already marked as present, the system proceeds to record the check-in by calling LogCheckIn(VID) through the **ActionLogger**, ensuring the event is documented in the system logs.

The sequence diagram demonstrates the use of software design patterns to coordinate interactions among objects. The Facade Pattern is evident in the **QRCodeManager**, which acts as a unified interface for the **Security Guard** to interact with the underlying subsystems, including **ValidationManager, RecordStorage,** and **ActionLogger**, thereby simplifying the process of QR code validation and check-in logging. The Chain of Responsibility Pattern is reflected in the flow from **QRCodeManager** to **ValidationManager** to **RecordStorage** during the ValidateScannedQR() operation, where the request is passed along a chain of handlers until it is processed. Lastly, the Command Pattern is employed in actions like LogCheckin(VID) where a request is encapsulated as an object, allowing it to be logged, tracked, or audited independently. These patterns promote separation of concerns, reusability, and scalability within the system architecture.

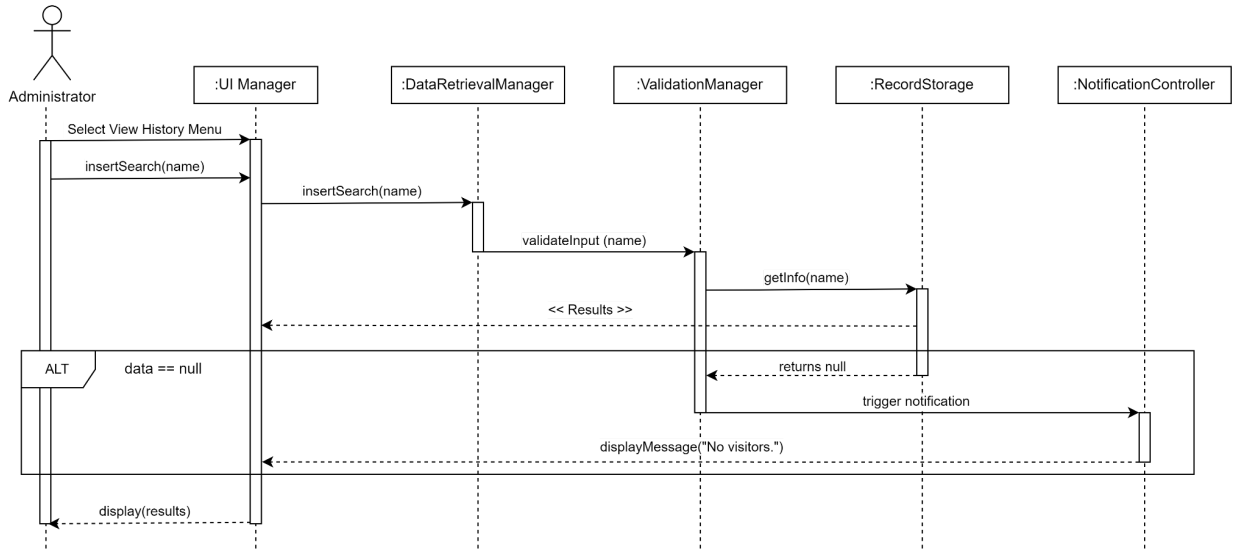

Figure 37.ViewHistory (UC-8)

The **Administrator** initiates the process by selecting the "View History" menu to search for visitor records. This action triggers the **UI Manager**, which sends an insertSearch(id, username) request to the **DataRetrievalManager** to begin the search. The **DataRetrievalManager** then calls the **ValidationManager** to check whether the provided ID and username are valid. If the input is correct, the **ValidationManager** proceeds by requesting visitor information from **RecordStorage** using getInfo(id).

Once **RecordStorage** processes the request, it returns the visitor records as results. These results are then sent back through the **DataRetrievalManager** to the **UI Manager**, which displays them to the administrator. However, if no visitor data is found, an alternate flow (ALT) is triggered. In this case, **RecordStorage** returns null, indicating that no records exist for the given ID. The **NotificationController** is then activated to trigger a notification, ensuring that the system

acknowledges the absence of visitor history. The **UI Manager** subsequently displays a message stating "No visitors," informing the administrator that no records match their search criteria.

The sequence diagram illustrates several software design patterns used in the visitor history lookup process. The Facade Pattern is applied through the **UI Manager,** which abstracts the complexity of underlying operations by serving as a single entry point for the **Administrator** to initiate a history search. The Chain of Responsibility Pattern appears in the validation flow, where the insertSearch(name) request propagates through **DataRetrievalManager** to **ValidationManager** and finally to **RecordStorage** via getInfo(name), with each component responsible for handling or passing on the request. The Strategy Pattern is suggested in the **ValidationManager**, which can employ various validation algorithms depending on the input type or context. The Observer Pattern is evident when the **NotificationController** is triggered to display a message (“No visitors.”), suggesting a reactive update mechanism based on system state changes. These patterns collectively ensure modularity, enhance code maintainability, and allow components to evolve independently while preserving functionality.

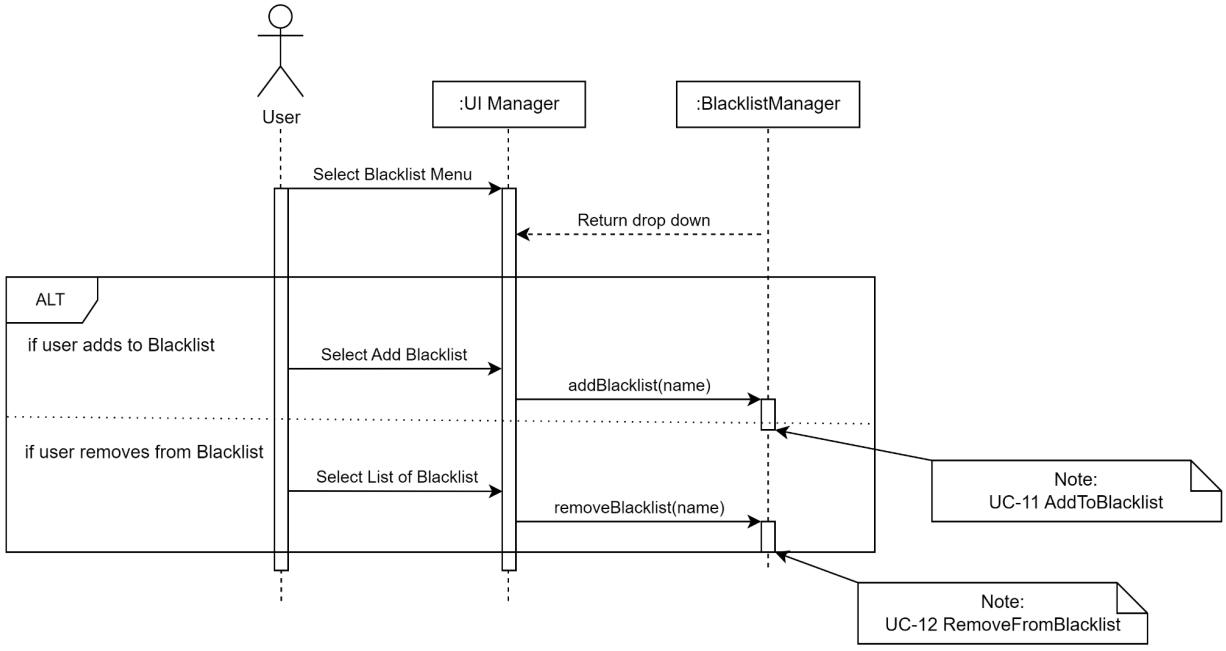

Figure 38. ManageBlacklist(UC-9)

In this interaction for managing the blacklist within the gated community system, the **User** begins by selecting the Blacklist Menu through the **UI Manager**, which responds by displaying a drop-down menu. This sequence is part of the general use case Manage Blacklist, which encompasses the specific actions of adding and removing individuals from the blacklist. If the user chooses to add someone, they select Add Blacklist, prompting the **UI Manager** to invoke addBlacklist(name) on the **BlacklistManager**, representing the functionality of UC-11 AddToBlacklist. If the user chooses to remove a person, they select List of Blacklist, which results in a removeBlacklist(name) request being sent to the **BlacklistManager**, aligning with

UC-12 RemoveFromBlacklist. These two conditional flows are represented using an ALT (alternative) fragment to show that either action can occur under the broader Manage Blacklist process.

The diagram demonstrates the use of software design patterns within the blacklist modification process. The Facade Pattern is applied through the **UI Manager,** which abstracts the complexities of blacklist operations by offering a simplified interface to the user for adding or removing entries. The Command Pattern is explicitly represented in the addBlacklist(name) and removeBlacklist(name) operations, where each action is treated as a command object encapsulating all the information needed for execution, allowing for traceability and potential rollback. Additionally, the Observer Pattern is through the user interaction flow where selecting an option from the dropdown dynamically influences the subsequent behavior of the system, indicating decoupled components that react to state changes. These patterns contribute to a clean, maintainable structure that supports flexible user interactions and backend processing.

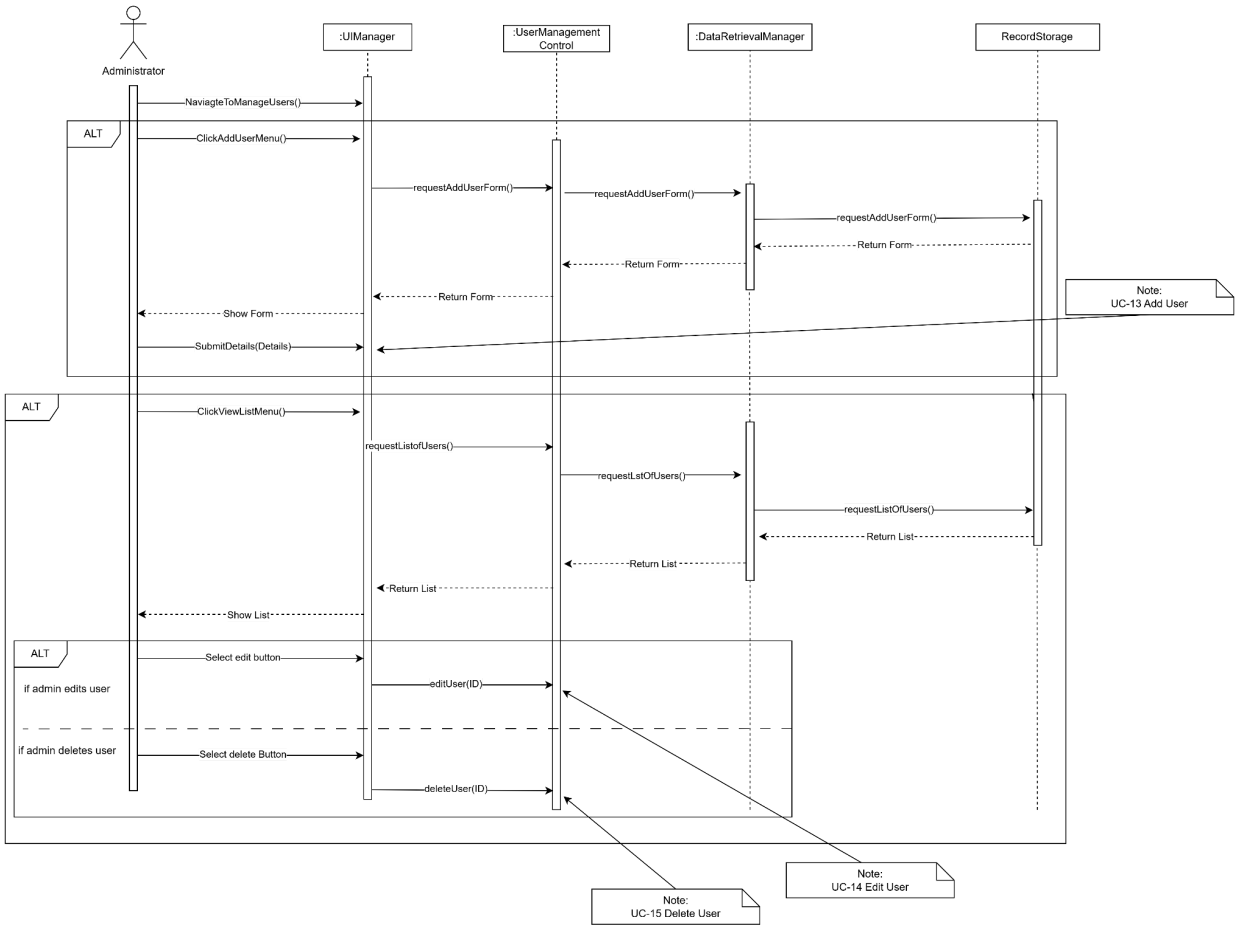

Figure 39.ManageUsers (UC-12)

The Administrator first navigates to the Manage Users interface by interacting with the **UIManager**. This action triggers an alternative (alt) flow where the Administrator can choose to either add a new user or view the list of existing users.

If the Administrator selects the Add User option, the **UIManager** sends a request to the **UserManagementController** (requestAddUserForm()). The **UserManagementController** then requests the **DataRetrievalManager** to retrieve the necessary form structure (requestAddUserForm()). The **DataRetrievalManager** retrieves the form from storage and returns it to the **UserManagementController**, which then passes it back to the UIManager. Once the form is displayed, the Administrator enters the user details and submits them (SubmitDetails(Details)).

If the Administrator selects the View List of Users option, the **UIManager** sends a request to the **UserManagementController** (requestListOfUsers()). The **UserManagementController** then requests the **DataRetrievalManager** to retrieve the list of users (requestListOfUsers()). The **DataRetrievalManager** fetches the list from the **RecordStorage** and returns it. The **UserManagementController** then sends the user list back to the **UIManager**, which displays it for the Administrator.

Once the user list is displayed, the Administrator has two options: edit a user or delete a user, which are handled within another alternative (alt) flow. If the Administrator selects the Edit User button, the **UIManager** sends an editUser(ID) request to the **UserManagementController**, which follows the **UC-14 Edit User** process. If the Administrator chooses to Delete User, the **UIManager** sends a deleteUser(ID) request to the **UserManagementController**, which follows the **UC-15 Delete User** process.

The interaction diagram demonstrates the application of software design patterns used in the user management workflow. The Facade Pattern is applied through the **UIManager**, which serves as a simplified interface for the **Administrator** to perform operations such as adding, editing, or deleting users, while encapsulating the complexities of the underlying logic handled by components like **UserManagementControl** and **RecordStorage**. The Command Pattern is clearly represented in the actions addUser, editUser, and deleteUser, where each operation is encapsulated as a discrete command, enabling parameterization, tracking, and potential rollback of requests. These patterns promote modularity, enhance maintainability, and provide a scalable foundation for managing user-related functionalities.

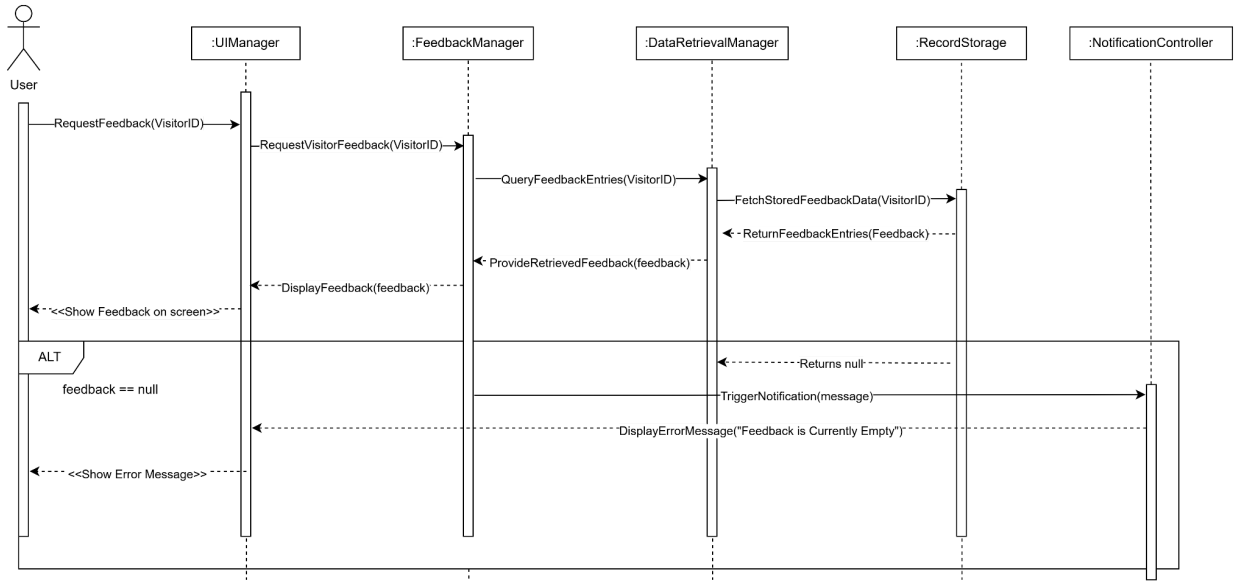

Figure 40. ViewFeedback (UC-16)

When a user wishes to view feedback associated with a particular visitor, the process begins with the user submitting a request via the interface. The **UIManager** receives this request through the **RequestFeedback(visitorID)** function and forwards it to the **FeedbackManager** using the **RequestVisitorFeedback(visitorID)** call.

The **FeedbackManager** initiates a query by calling **QueryFeedbackEntries(visitorID)** on the **DataRetrievalManager**, which is responsible for searching stored information. Upon receiving the query, the **DataRetrievalManager** interacts with the **RecordStorage** component to fetch the relevant feedback data by executing **FetchStoredFeedbackData(visitorID)**.

Once the data is retrieved, the **RecordStorage** returns the feedback entries to the **DataRetrievalManager**, which then sends the results back to the **FeedbackManager** using **ProvideRetrievedFeedback(feedback)**. Finally, the **FeedbackManager** returns the feedback data to the **UIManager**, which displays the information on the screen for the user.

If no feedback is found, the system follows an alternate flow. The **FeedbackManager** triggers a notification using **TriggerNotification(message)** directed to the **NotificationController**. The **NotificationController** then displays the appropriate message using **DisplayErrorMessage(“Feedback is Currently Empty”)**, informing the user of the issue or lack of data.

The interaction diagram demonstrates use of software design patterns in the feedback retrieval process. The Facade Pattern is represented by the **UIManager**, which simplifies user interaction by managing requests and responses while hiding internal complexities. The Chain of Responsibility Pattern appears in the flow from **FeedbackManager** to **DataRetrievalManager** to **RecordStorage**, where each component handles part of the feedback query. The Observer Pattern is shown when **FeedbackManager** triggers **NotificationController** to display a message

if no feedback is found. These patterns promote modularity, clarity, and efficient communication in the system

Class Diagram and Interface Specification

Class Diagram

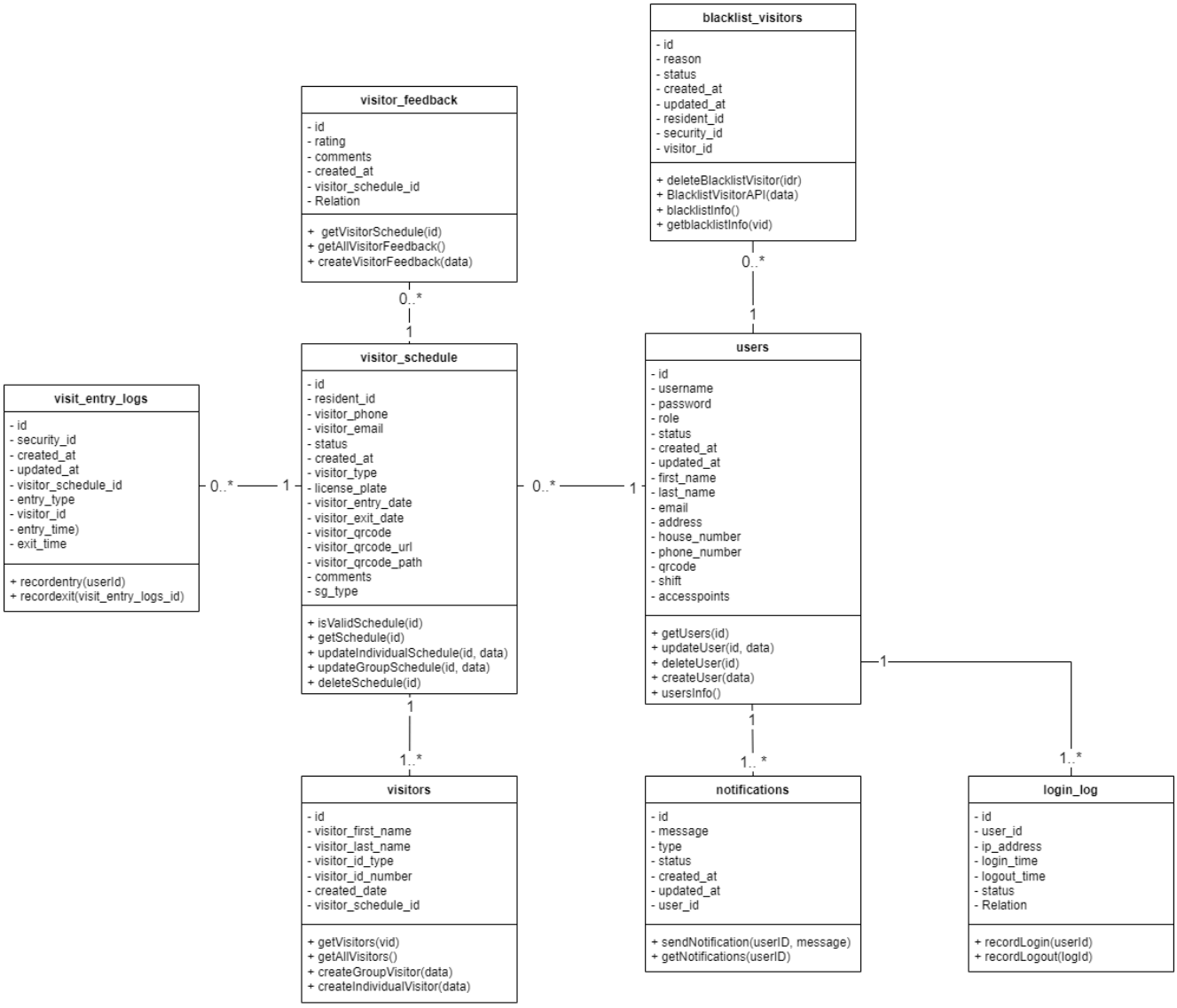

Figure 41. Class Diagram

Design Patterns

The visitor management system utilities several well-established software design patterns categorized into Structural, Creational, and Behavioral patterns. These patterns help organize the system's logic, support reusability, and enhance maintainability.

1. Structural Design Patterns

Active Record Pattern

Each entity in the system (e.g., users, visitor_schedule, visit_entry_logs) follows the Active Record pattern by encapsulating both data and behavior. For instance, the users table includes methods such as getUsers(id) and createUser(data) that operate directly on its attributes. This design simplifies database operations by coupling data access logic with the data model.

Repository Pattern

The system uses repository-like methods such as getAllVisitors(), getSchedule(id), and getNotifications(userID) to abstract data access operations. This pattern separates the business logic from data access logic, enabling better scalability and testability.

2. Creational Design Patterns

Factory Method Pattern

The system utilizes factory methods for creating new objects. For example, methods like createUser(data) and createGroupVisitor(data) encapsulate the instantiation logic for different entities. This approach allows object creation to be centralized and consistent throughout the application.

3. Behavioral Design Patterns

Observer Pattern

The notifications component demonstrates the Observer pattern. Methods such as sendNotification(userID, message) and getNotifications(userID) allow the system to notify users when specific events occur (e.g., visitor check-in), promoting loose coupling between event producers and subscribers.

Object Constraint Language Contract

Contract Name:	**CreateVisitorSchedule()**
Cross Reference	Resident, Visitor, QRCode, Notifications
Invariants:	Resident is logged in Visitor information must be valid Visitor registration is associated with a defined schedule
Precondition	Resident must be logged into the system
Post Condition	Visitor is successfully registered in the system QR Code is generated and sent to the visitor

Table 43. OCL Contract for CreateVisitorSchedule

Contract Name:	**SubmitFeedback()**
Cross Reference	Visitor, Database, QRCode
Invariants:	Visitor has previously checked in Visitor identity is verifiable via QR code
Precondition	System has stored visitor check-in data
Post Condition	Feedback is successfully submitted and stored in the database

Table 44. OCL Contract for SubmitFeedback

Contract Name:	**ScanQR()**
Cross Reference	Security Guard, Visitor, QRCode
Invariants:	Visitor is registered with a valid QR code Security guard is authenticated and authorized to use the system
Precondition	Visitor must be registered in the system with a valid QR code Security guard must have access to the check-in system

Post Condition	Visitor is successfully checked in Entry is granted or denied based on QR validation

Table 45. OCL Contract for ScanQR

Contract Name:	**ViewHistory()**
Cross Reference	Security Guard, Database
Invariants:	Security guard is authenticated Visitor check-in data is available in the system
Precondition	Security guard is logged into the system Visitor check-in data exists in the database
Post Condition	Visitor history is retrieved and displayed to the security guard

Table 46. OCL Contract for ViewHistory

Contract Name:	**ManageBlacklist()**
Cross Reference	Resident, Security Guard, Database
Invariants:	User must be logged into the system Visitor must have an existing record
Precondition	Resident or Security Guard is logged in Visitor record exists in the system
Post Condition	Visitor is added to or removed from the blacklist

Table 47. OCL Contract for ManageBlacklist

Contract Name:	**ManageUsers()**
Cross Reference	Administrator, Database
Invariants:	Administrator must be authenticated User records are uniquely identifiable

Precondition	Administrator is logged into the system User records exist in the database
Post Condition	A user account is added, edited, or deleted from the system

Table 48. OCL Contract for ManageUsers

Contract Name:	**ViewFeedback()**
Cross Reference	Administrator, Resident, Security Guard, Database
Invariants:	User must be logged in
Precondition	Administrator, Resident, or Security Guard is logged into the system At least one feedback entry exists in the database
Post Condition	Visitor feedback is successfully displayed to the authorized user

Table 49. OCL Contract for CreateVisitorSchedule

Data Types and Operation Signatures

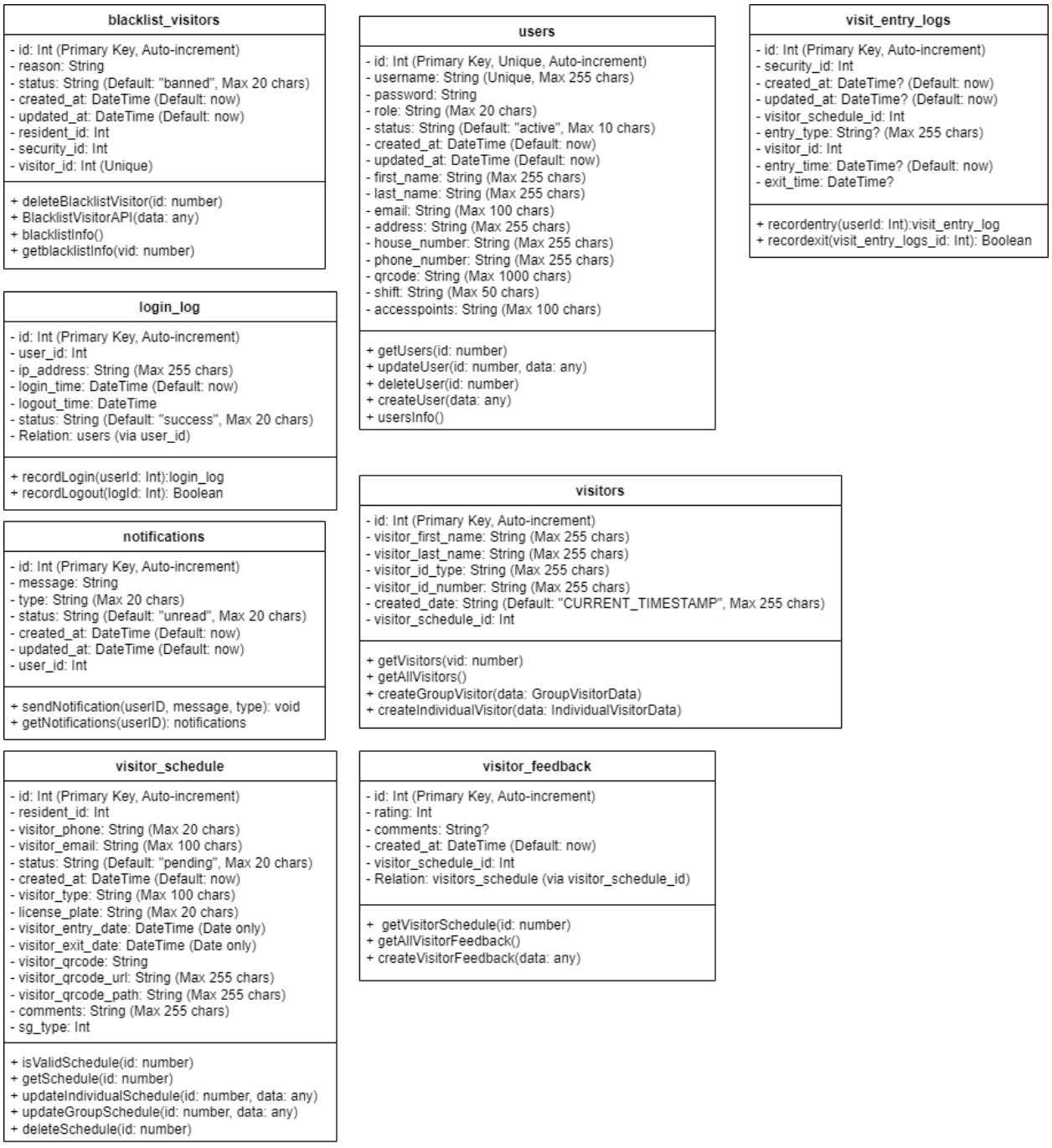

Figure 42. Data Types and Operation Diagram

Traceability Matrix

	Login Log	Notification	Users	Blacklist Visitors	Visitor Feedback	Visitors	Visitor Schedule	Visitor Entry Log
UI Manager				✅			✅	
Controller				✅		✅		
Record Storage	✅							✅
QRCode Manager						✅		
Notification Controller		✅						
Visitors						✅		
Visit Manager							✅	
Feedback Manager					✅			
Validation Manager	✅		✅			✅		✅
Action Logger	✅							✅
Data Retrieval Manager				✅	✅			
Blacklist Manager				✅				
User Management Controller			✅					

Table 50. Traceability Matrix of Classes and Concept

Algorithms and Data Structures

Algorithms

We've developed the algorithm to enhance real-time user tracking through our entrance system. This algorithm is used to verify whether a visitor is a one-time guest, a recurring visitor, or blacklisted from entering the gated community. This system integrates the use of QR codes for automated access and manual requests for entry, ensuring a seamless and secure process. When a visitor arrives, their information is captured and checked against the database. If the visitor is blacklisted, access is denied, and the attempt is logged. If the visitor is already in the system as a recurring guest, their information is verified, and access is granted. If the visitor is new, the system validates their data before granting access and saving their information for future visits. All entry attempts are logged to ensure accountability and maintain a proper check-and-balance process.

Algorithm Design		
Step	Title	Description
1	Initialize System	Import necessary libraries (e.g., database connectors, QR code readers, graph plotting tools).Connect to the database that stores user data, visitor schedules, entry logs, and blacklist information.
2	User Entry Process	**Input Validation:** Check if the user is using a QR code or manual entry request. **For QR code:** Scan the QR code and extract user/visitor information (e.g., VisitorID, AccessTime). Validate QR code (e.g., check if it's expired or already used). **For manual entry:** Verify user details against the database (e.g., VisitorID, Name, Contact Details). **Blacklist Check:**Query the database to confirm if the user/visitor is blacklisted. **If blacklisted:** Deny entry and log the attempt with details (time, reason for denial) and notify security personnel via the dashboard. **Access Authorization:** If the user passes validation: Update visitor status to "Active" in the database. Log entry details (VisitorID, EntryTime, EntryType).
3	Real-Time Dashboard Update	**Active Attendees:** - Count users currently in the community by querying "Active" status in the visitor logs. - Generate and display real-time graphs showing the number of active attendees. **Blacklisted Attendees:** - Query the blacklist records and display the count along with details (VisitorID, Name, Reason). - Update graphs dynamically using streaming data every few seconds.
4	Notifications and Alerts	- Trigger notifications for any abnormal activities (e.g., unauthorized entry attempts). - If a blacklisted visitor tries to gain access: - Send an alert to security personnel and notify the associated resident.
5	Exit Process	Update visitor status to "Inactive" in the database and log exit details (VisitorID, ExitTime).
6	Generate Reports	- Allow administrators to view detailed reports via the dashboard: - Daily active attendee counts. - Entry attempts and blacklist updates.

Table 51. Algorithm Design Descriptions

Data Structures

Our system uses arrays to manage and store visitor information due to their simplicity and ability to represent one-to-many relationships effectively. For example, a single resident can have multiple approved visitors, and an array allows us to group and access this related data efficiently. We chose arrays because they offer fast access to elements by index, which improves performance when checking or updating visitor lists. While more complex structures like hash tables offer additional flexibility, arrays provide the right balance of performance and ease of implementation for our system's needs.

Representation of One-to-Many Relationships: Arrays are ideal for grouping multiple approved visitors under a single resident. For example, each resident can have an array of approved visitor IDs, making it straightforward to fetch, add, or remove specific visitors related to a resident.

Fast Index-Based Access: Arrays provide constant time complexity (O(1)) for accessing elements by index. This is particularly useful for checking visitor permissions quickly during gate access validation, ensuring smooth operations without delays.

Ease of Traversal and Updates: Whether we are adding a new visitor, removing one, or updating their status, arrays allow simple and efficient traversal and modification. Looping through an array to check for a specific visitor ID or update their details is intuitive and quick.

Memory Efficiency: Compared to complex structures, arrays typically require less memory and are a good fit for systems where high performance and minimal overhead are priorities.

Scalability Challenges: While arrays excel in many areas, it is worth considering their limitations for scalability. As the number of visitors grows, operations like searching for a

specific visitor or removing an entry might become less efficient. If scalability becomes a concern, hybrid approaches combining arrays with hash tables or trees could be explored.

Integration with QR Code System: Arrays can efficiently store generated QR codes mapped to visitor IDs, enabling quick validation during entry.

User Interface Design and Implementation

Resident- UC-2 CreateVisitor Schedule

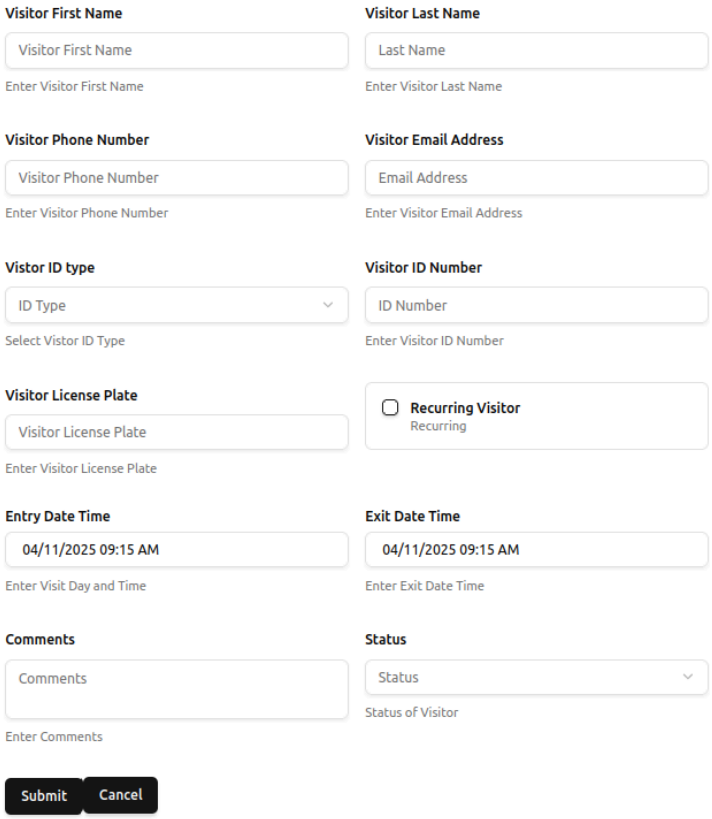

Figure 43.Create individual visit schedules

The Create Visitor Schedule use case was modified to enhance user-friendliness. Users can now select entry and exit times by simply clicking on the date field and choosing the desired date and time from a picker.

For the Visitor ID Type (Social Security, Passport, or Driver's License) and Status (Active or Inactive), users can select from dropdown menus, making it easier for them. To mark a visitor as recurring, the user can click a checkbox labeled accordingly.

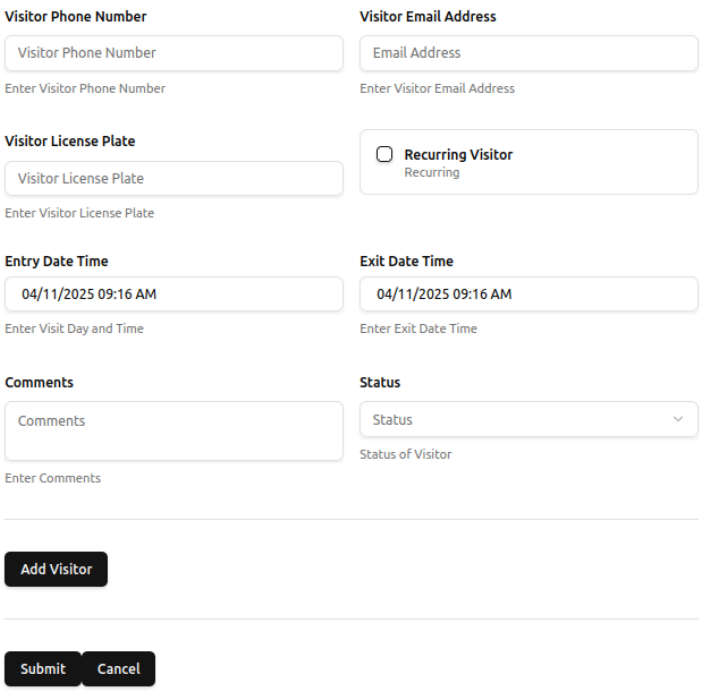

Figure 44. Create group visit schedules

In the Group Visit Schedule, if users wish to add another visitor to the group, they can click the Add Visitor button, which will allow them to enter the new visitor's information.

UC-8 ViewHistory

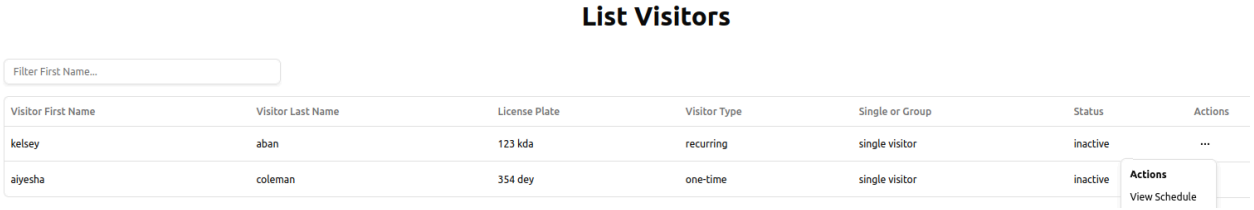

Figure 45. View List of visitors

The View History feature was updated to improve user experience. Users can now click on the three-dot menu beside each visitor entry to access additional actions

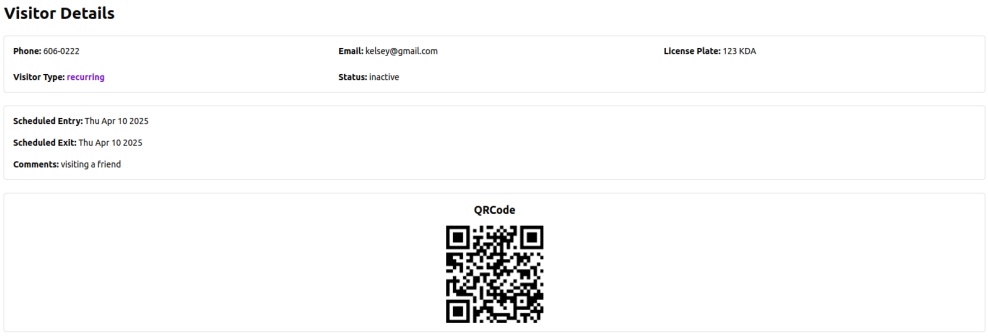

Figure 46. View visitors details

By selecting "View Schedule" under Actions, they can view the visitor's full schedule in more detail. This helps keep the main visitor list clean and uncluttered, while still providing easy access to detailed information when needed.

UC-9 ManageBlacklist

Figure 47. Manage Blacklist Main

The Manage Blacklist feature works similarly to the View History function. Users can view the existing list of visitors and, if they wish to blacklist a visitor, they can click on the three-dot menu next to the visitor's name and select "Add to Blacklist."

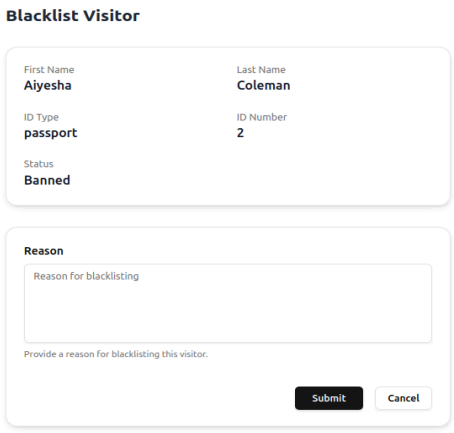

Figure 48.Submit Reason for Blacklist (Form)

Upon selection, a form will appear prompting the user to provide a reason for blacklisting the visitor. After entering the reason, the user can choose to Submit the form to finalize the action or click Cancel to return to the visitor list without making any changes. This makes the process more user-friendly, as it gives users the flexibility to cancel the action easily, especially helpful if they selected the wrong visitor by mistake.

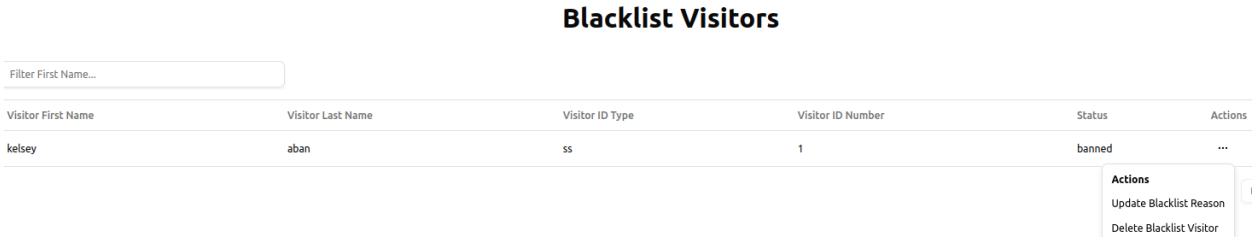

Figure 49. Blacklisted visitors

To remove a visitor from the blacklist, the user can click on "Delete Blacklisted Visitor," which will prompt a confirmation message. The user can then choose OK to proceed or Cancel to stop the action. If the user wants to update the reason for blacklisting, they can do so easily by selecting "Update Blacklist Reason" under the Actions menu.

The application is designed to be user-friendly, allowing users to navigate smoothly through various features without confusion. Some layout adjustments were made specifically to enhance usability and minimize user effort. For example, features like the three-dot menu keep the interface clean by grouping actions (such as viewing schedules or managing the blacklist) in a compact, intuitive way. This reduces visual clutter and makes it easier for users to focus on the task at hand.

The interface follows a minimalist design, avoiding unnecessary graphics, bright colors, or distracting animations. Instead, it emphasizes clarity and functionality. Key actions such as adding to the blacklist, updating reasons, or deleting entries are clearly labeled and accessible with just a few clicks. The use of forms with straightforward input fields, accompanied by Cancel and Submit buttons, ensures users can complete or exit a task without frustration.

Additionally, commonly used functions are placed in predictable and consistent locations, so users don't need to guess or refer to external instructions. This aligns with usability principles such as consistency, visibility of system status, and user control and freedom, all of which contribute to a seamless and intuitive user experience.

Design of Tests

Unit testing will focus on individual components to ensure they perform as expected. The following test cases will be programmed:

Test-case Identifier:	TC -1	
Use Case Tested:	**UC-2 CreateVisitSchedule**, UC-3 Generate QR Code, UC-5 SendNotifications and UC-4 SendQRCode	
Pass/fail Criteria:	The test is successful if the visit schedule is successfully created and the response is a 200 status code.	
Input Data:	Visitor details, schedule date, and time.	
Test Procedure:		**Expected Data:**
1. Connect to Database 2. Call function validateScheduleInput(input) with valid visitor and date info. 3. Call function createVisitSchedule(userId, input) with valid user ID. 4. Call function notifyUser(userId)		 Success (200) Success (200) Success (200)

Table 52. Test Case for CreateVisitSchedule (UC-2)

This test case covers the workflow for creating a visit schedule. It validates visitor details, date, and time resulting in a successful schedule creation, QR code generation, and proper notifications to residents. All test functions should return a successful server response (200), confirming the system handles visit scheduling end-to-end without errors.

Test-case Identifier:	TC -2	
Use Case Tested:	**UC- SubmitFeedback**	
Pass/fail Criteria:	The test is successful if the feedback is submitted and stored successfully.	
Input Data:	Alphanumeric(userID, comment text, rating value)	
Test Procedure:		**Expected Data:**
1. Create connection with database 2. Call function validateFeedback(input) with valid feedback text and rating. 3. Call function validateFeedback(input) with empty feedback. 4. Call function validateFeedback(input) with invalid rating (e.g. 6). 5. Call function submitFeedback(userId, feedback) with valid user ID and feedback. 6. Call function submitFeedback(userId, feedback) with non-existent user ID.		Success (200) Failure (400) Failure (400) Success (200) Success (404)

Table 53. Test Case for SubmitFeedback (UC-6)

This test addresses UC-6 (SubmitFeedback) and covers multiple validation scenarios. It tests whether valid feedback with appropriate text and rating is successfully submitted, while also verifying that invalid inputs such as empty comments, out-of-range ratings, or non-existent user IDs are correctly rejected. The system's validation logic and submission mechanism are confirmed through appropriate server responses (200 for success, 400 and 404 for errors).

Test-case Identifier:	TC - 3	
Use Case Tested:	UC-7 ScanQR	
Pass/fail Criteria:	The test is successful if the QR scan returns valid visitor data and the response is a 200 status code.	
Input Data:	Valid QR code.	
Test Procedure:		**Expected Data:**
1. Open QR Scanner. 2. Call function decodeQR(code). 3. Call function validateQRCodeData(data) 4. Call function getVisitors(visitorId) 5. Call function markAsScanned(visitorId) 6. Call function checkBlacklist(visitorId)		Success Success (200) Success (200) Success (200) Success (200) Success (200)

Table 54. Test Case for ScanQR(UC-7)

Covering UC-7 (ScanQR), this test ensures that the QR scanning mechanism works correctly. It validates that a QR code can be decoded, its data can be checked for correctness, and the associated visitor can be retrieved and marked as scanned. It also confirms that the system checks for blacklist status before allowing access. All processes return 200 status codes, indicating a complete and successful scan and verification workflow.

Test-case Identifier:	TC - 4	
Use Case Tested:	**UC-8 ViewHistory**	
Pass/fail Criteria:	The test is successful if the user can view the history of their schedules and the response is a 200 status code.	
Input Data:	Resident ID.	
Test Procedure:		**Expected Data:**
1. Create connection with database 2. Call function getVisitorHistory(userId) 3. Call function renderHistory(data) 4. Call function getVisitorHistory(userId) with invalid user ID		Success Success (200) Success (200) Failure (404)

Table 55. ViewHistory (UC-8)

This test corresponds to UC-8 (ViewHistory) and verifies the retrieval of a resident's visit history. It confirms successful data retrieval for valid user IDs and proper error handling for invalid ones. The rendering function is also tested to ensure correct output display. A successful path returns a 200 code, while the invalid user case is properly rejected with a 404.

ManageBlacklist(UC-9) includes 2 other use cases: UC-10 Add to Blacklist and UC-11 Remove from Blacklist. Hence, there are multiple test cases that were implemented for this general use case.

Test-case Identifier:	TC - 5	
Use Case Tested:	**UC-10 AddtoBlacklist**	
Pass/fail Criteria:	The test is successful if the visitor is successfully added to the blacklist and the response is a 200 status code.	
Input Data:	Visitor ID, blacklist reason	
Test Procedure:		**Expected Data:**
1. Create connection with database 2. Call function addToBlacklist(visitorId, reason) 3. Call function notifyAdmins(visitorId)		Success Success (200) Success (200)

Table 56. Test Case for AddtoBlacklist (UC-10)

This test case focuses on UC-10 (AddtoBlacklist) and verifies that a visitor can be added to the blacklist using a valid ID and reason. It also checks that appropriate notifications are sent to administrators after blacklisting. The server returns a 200 status code for each step, confirming the blacklist addition process works as expected.

Test-case Identifier:	TC - 6	
Use Case Tested:	**UC-11 RemovefromBlacklist**	
Pass/fail Criteria:	The test is successful if the visitor is successfully removed from the blacklist and the response is a 200 status code.	
Input Data:	Visitor ID	
Test Procedure:		**Expected Data:**
1. Create connection with database 2. Call function checkBlacklistStatus(visitorId) 3. Call function removeFromBlacklist(visitorId)		Success Success (200) Success (200)

Table 57. Test Case for RemovefromBlacklist (UC-11)

Linked to UC-11 (RemovefromBlacklist), this test validates that the system correctly identifies a blacklisted visitor and removes them from the list upon request. Successful database connection and removal processes yield a 200 response, confirming the system can update blacklist status reliably.

ManageUsers (UC-12) includes 3 other use cases: UC-13 AddUser, UC-14 EditUser and UC-15 Delete User. Hence, there are multiple test cases that were implemented for this general use case.

Test-case Identifier:	TC - 7	
Use Case Tested:	**UC- 13 AddUser**	
Pass/fail Criteria:	Users are successfully added to the system.	
Input Data:	Name, email, password, role	
Test Procedure:		**Expected Data:**
1. Connect to the database. 2. Call function validateUserInput(input) with valid name, email, and role. 3. Call function validateUserInput(input) with empty name. 4. Call function checkDuplicateUser(email) with the same email. 5. Call function checkDuplicateUser(email) with existing email. 6. Call function createUser(input) with new user data. 7. Call function createUser(input) with invalid role.		Success Success (200) Failure (400) Success (200) Failure (400) Failure (400) Success (200)

Table 58. Test Case for AddUser (UC-13)

Covering UC-13 (AddUser), this test ensures the system can register new users using valid name, email, and role inputs. It checks for validation errors when required fields are empty or when duplicate emails are used, and it also tests the rejection of invalid role values. The test confirms that successful registrations receive a 200 response, while input or logical errors return 400.

Test-case Identifier:	TC - 8	
Use Case Tested:	**UC-EditUser**	
Pass/fail Criteria:	The test is successful if the user is successfully updated and the response is a 200 status code.	
Input Data:	User ID, updated details (e.g., password, username, role).	
Test Procedure:		**Expected Data:**
1. Connect to the database 2. Call function getUserById(userId) 3. Call function updateUser(userId, newData)		Success Success (200) Success (200)

Table 59. Test Case for EditUser (UC-13)

This test supports UC-14 (EditUser) and covers the flow of retrieving a user by ID and updating their information. It verifies that the system accepts valid updates such as new usernames, passwords, or roles. Successful updates return a 200 status code, demonstrating reliable user data modification.

Test-case Identifier:	TC - 9
Use Case Tested:	**UC-13 DeleteUser**
Pass/fail Criteria:	The test is successful if the user is successfully deleted and the response is a 200 status code.
Input Data:	User ID

Test Procedure:	**Expected Data:**
1. Connect to the database 2. Call function getUserById(userId) 3. Call function getUserById(userId) with no user found 4. Call function deleteUser(userId)	Success Success(200) Failure(400) Success(200)

Table 60. Test Case for DeleteUser (UC-13)

Aligned with UC-15 (DeleteUser), this test ensures the system can remove a user account. It confirms that an existing user can be found and deleted, and that attempts to delete a non-existent user are handled gracefully. A 200 status is returned for successful deletions, while failure to locate the user results in a 400 response.

Test-case Identifier:	TC - 10
Use Case Tested:	**UC- 16 ViewFeedback**
Pass/fail Criteria:	The test is successful if the feedback for a visitor schedule is successfully viewed and the response is a 200 status code.
Input Data:	Schedule ID.

Test Procedure:	**Expected Data:**
1. Connect to the database 2. Call function getUserById(userId) 3. Call function updateUser(userId, newData) 4. Call function getAllVisitorFeedback() 5. Call function renderFeedback(data) 6. Call function renderFeedback(data) with empty database	Success Success (200) Success (200) Success (200) Success (200) Failure (400)

Table 61.Test Case for ViewFeedback (UC-16)

This test case maps to UC-16 (ViewFeedback) and checks whether the system can retrieve and render feedback for visitor schedules. It verifies the retrieval of all feedback entries and checks how the system behaves when no feedback exists. Server responses include 200 for successful retrieval and 400 when the feedback list is empty, validating both typical and edge-case behavior.

Integration Testing Strategy

The integration testing strategy will focus on verifying seamless interaction between different modules:

1. **Testing CRUD Integration with Database:** Ensure visitor data flows correctly between the app and database, and verify data remains consistent during simultaneous CRUD operations.
2. **QR Code Integration:** Test full flow from QR code generation to gate validation, ensuring proper error handling for invalid codes.
3. **Real-Time Dashboard Updates:** Simulate entry/exit events to verify real-time dashboard updates and synchronization with entry logs.
4. **Integration of Blacklist Module:** Ensure blacklisted visitors are denied access and dashboard reflects blacklist updates instantly.
5. **System-wide Workflow Testing:** Simulate full user workflows (e.g., scheduling visits, generating QR codes, viewing dashboard) to confirm seamless integration across all modules.

Plan for Algorithm, Non-functional, and UI Testing

1. **Algorithm Testing:** Run unit tests to verify accuracy of validation algorithms using mock data and edge cases like expired QR codes or blacklisted users.
2. **Non-functional Testing:** Evaluate system performance under heavy load and confirm proper data security measures like encryption are in place.
3. **UI Testing:** Check user interface for ease of navigation, form usability, and real-time updates; ensure clear and helpful error messages.

Project Management

Our team has been working on a Gated Community Application that facilitates secure visitor scheduling, access control, and community management. The goal is to provide residents and security personnel with an efficient way to manage visitors through features like scheduling, QR scanning, and blacklist management.

We use Miro, Jira, and GitHub to streamline project management, support collaboration, and ensure all tasks are completed efficiently and on time. Miro provides a visual overview of the project timeline and key milestones, helping the team stay aligned and identify potential delays early. Jira is used to break down tasks into manageable units, assign them to the appropriate team members, and monitor progress in real time. It also supports sprint planning, which helps us stay organized and adapt priorities as needed. For development, GitHub and Git allow us to work on different branches, track changes, and merge code seamlessly. This setup ensures proper version control, reduces conflicts, and supports smooth collaboration among developers. By combining these tools, we maintain a clear workflow, stay on schedule, and promote accountability throughout the project's lifecycle.

Development involved coordination among members, integration of various modules, and the use of shared repositories and communication channels to manage tasks. While team members faced challenges with balancing work and academic commitments, regular meetings increased our progress, clarified technical responsibilities, and addressed issues efficiently. Verbal updates during meetings proved more effective than relying solely on text-based communication.

History of Work

Implemented Use Cases:

- UC-2: Create Visitor Schedule
- UC-6: Submit Feedback
- UC-8: View History
- UC-10: Add to Blacklist
- UC-11: Remove from Blacklist

- UC-13: Add/Edit/Delete User
- UC-16: View Feedback
- Front end UC-7: Scan QR Code
- Additional backend refinements and UI polishing

Several key milestones have already been achieved in the project timeline. The initial report (Report #1) was completed by early March. System Architecture was finalized by the end of February, laying the groundwork for core development. Coding and UI Development started in early March, with substantial progress made since then. Debugging also commenced in parallel, continuing into April. Report #2, our current report, was completed around mid-April, documents between late March and mid-April. The first demo (Demo #1) took place was delivered along with Report #2 in April, marking a major milestone in showcasing progress. As of now, Report #3 and Demo #2 will be delivered on May 22, 2025.

Project Goals:

- Complete system architecture design
- Implement core coding functionalities
- Conduct initial testing and debugging
- Deliver Report #1 and Report #2
- Host Demo #1 showcasing system features
- Finalize and polish UI development
- Continue debugging and stability improvements
- Deliver Report #3 and conduct Demo #2
- Complete Application Interface and E-Archive module
- Deploy system for public or internal access
- Ensure system is user-friendly and functional end-to-end

APIs were developed for key functionalities including managing blacklists, feedback, users, and visitors. Each team member was assigned a specific API to develop, code, and test. Our lead programmer coordinated the integration of these APIs, ensuring that they worked seamlessly together. Interaction testing was carried out based on the API model that was initially designed. While each team member had specific responsibilities, the team collaborated and assisted each

other whenever needed to ensure smooth progress and resolve any challenges that arose during development.

Future development

Future goals may include enhancing user interaction between residents, administrators, and security personnel to foster better communication, improve responsiveness, and strengthen overall security. By encouraging active involvement and feedback, the system aims to create a safer, more connected, and confident community environment.

- **Resident Chat Support:** Allow residents to chat with guards or administrators within the app.
- **Issue Reporting Module:** Allow users to report bugs or issues directly from the app.

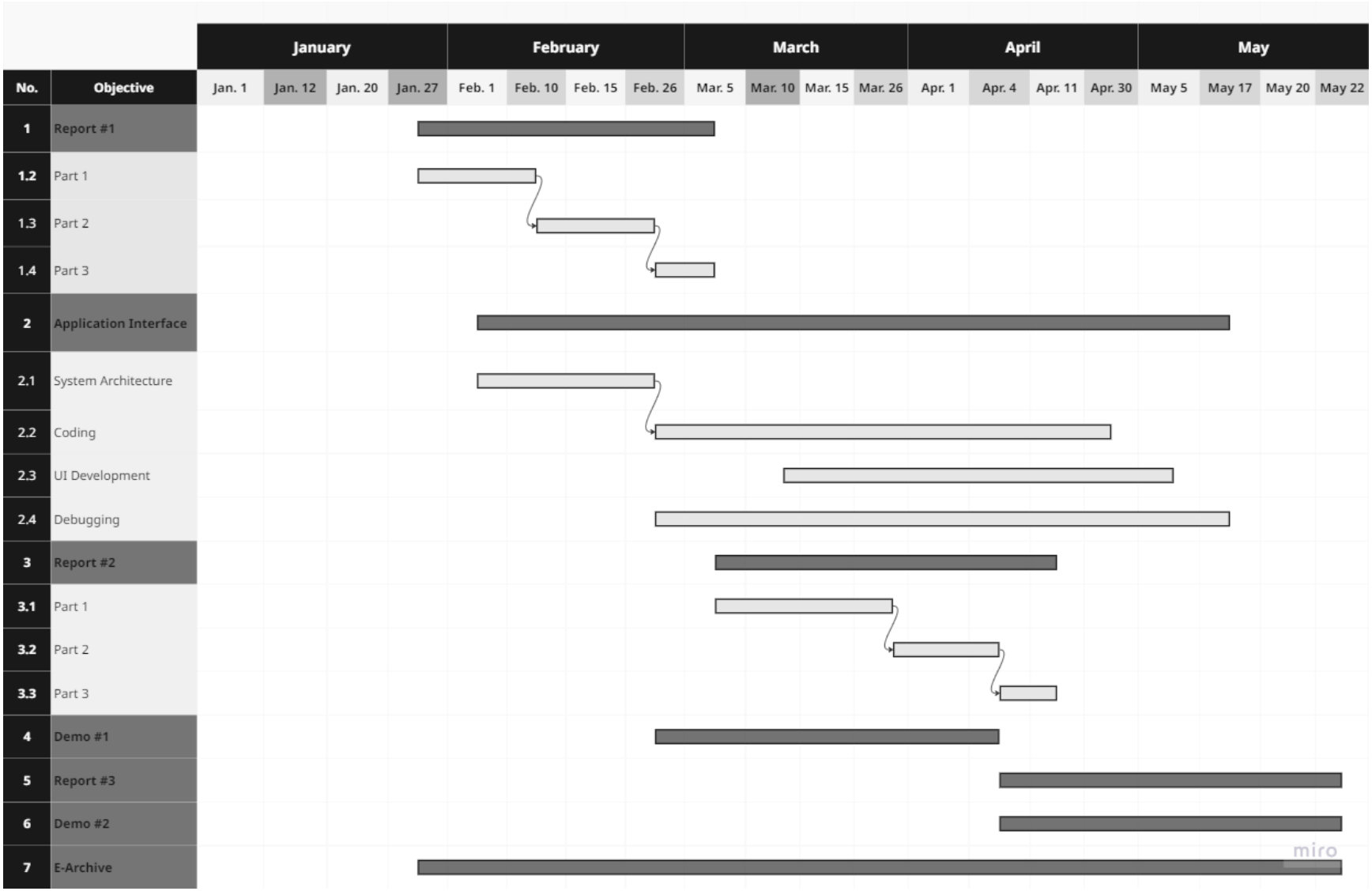

Figure 50. Gantt Chart

Summary of Changes

As the project progressed through multiple iterations, several adjustments were made to improve the clarity, consistency, and accuracy of the system design. The following key changes were implemented across the documentation and models:

- Refined the non-functional requirements to make them more measurable, specific, and clearly aligned with system expectations.
- Corrected use case descriptions to resolve inconsistencies in numbering and ensure alignment with system behavior.
- Updated sequence diagrams by generalizing the representation of processes for clarity. Scan QR, View History, and Manage Blacklist sequence diagram and interaction diagram was revised to reflect our present workflows.
- Simplified subsystem design by organizing them based on our projects major use cases, enhancing clarity and reducing unnecessary complexity.
- Removed Vehicle and Entry Log from the class diagram and documentation due to their exclusion from the final system implementation.

References

PalAmerican Security.(2021, August 10). *6 Security Procedures Your Gated Community Needs.* PalAmerican Security. https://www.palamerican.com/community/6-security-procedures-your-gated-community-needs/#_kx1a848liam6

Hope, H., Tzib, E., & Shol, J. (2024, December 8). *SSP5: Gated community management system.* Retrieved from https://sites.google.com/ub.edu.bz/gated-community-management-app/home/ssp5