

University of Belize



Report #2: Gated Community Application

Group #1

Kelsey A., Jordani A., Arthur B., Aiysha C., Aiden P.

University of Belize

CMPS4131- Software Engineering

Mr. Medina Manuel

April 11 2025

Table of Content

Analysis and Domain Modeling.....	3
Concept Model.....	3
I. Concept Definitions.....	3
II. Association Definitions.....	7
III. Attribute Definitions.....	10
IV. Traceability Matrix.....	11
V. Domain Model.....	12
System Operation Contracts.....	17
Data Model & Persistent Data Storage.....	22
Mathematical Model.....	24
Interaction Diagrams.....	25
Class Diagram and Interface Specification.....	33
Class Diagram.....	33
Data Types and Operation Signatures.....	34
Traceability Matrix.....	36
Algorithms and Data Structures.....	37
Algorithms.....	37
Data Structures.....	39
User Interface Design and Implementation.....	40
Design of Tests.....	44
Integration Testing Strategy.....	50
Plan for Algorithm, Non-functional, and UI Testing.....	50
Project Management.....	51
References.....	55

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 1. 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 2. 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
Notify residents upon visitor arrival.	D	NotificationController
Log all visitor check-in attempts.	D	ActionLogger

Table 3. 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 4. Concept Definition for ViewHistory (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 5. 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 6. 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 7. 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 8. 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 9. 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
ValidationManager ↔ NotificationController	QRCodeManager notifies residents upon visitor arrival.	Sends arrival notification
QRCodeManager ↔ ActionLogger	QRCodeManager logs all visitor check-in attempts.	Logs check-in

Table 10. 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 11. 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 12. 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 13. 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 14. 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, QRCode, QRCodeURL, QRCodePath, ExpirationDate	Manages the generation and validation 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 15. General Attribute Definitions

IV. Traceability Matrix

Use Case	P W	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			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 16. Table Showing Domain Analysis Traceability Matrix

V. Domain Model

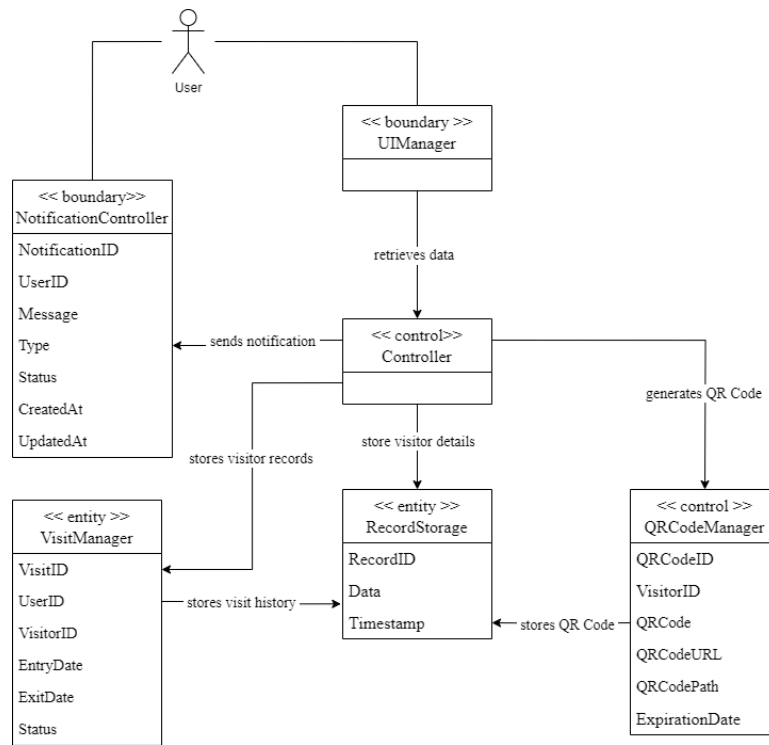


Figure 1. Domain Model for CreateVisitSchedule (UC-2)

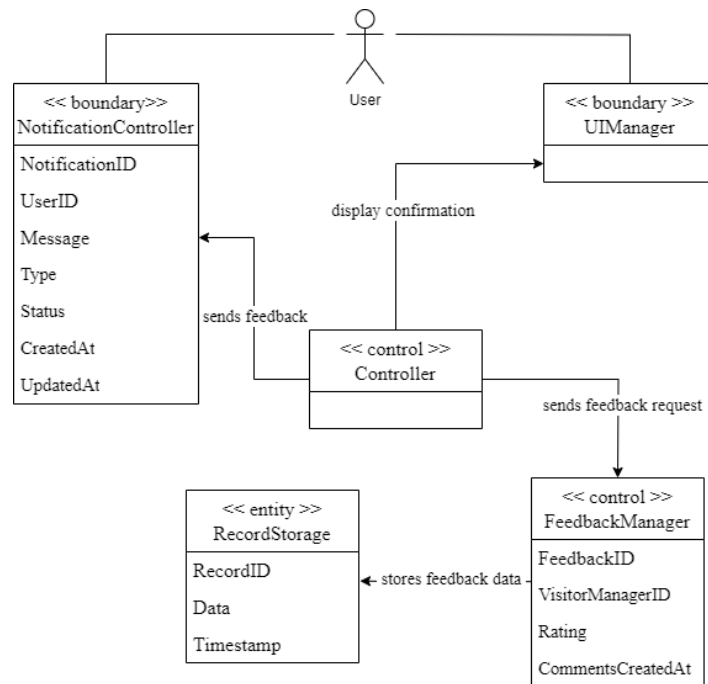


Figure 2. Domain Model for SubmitFeedback (UC-6)

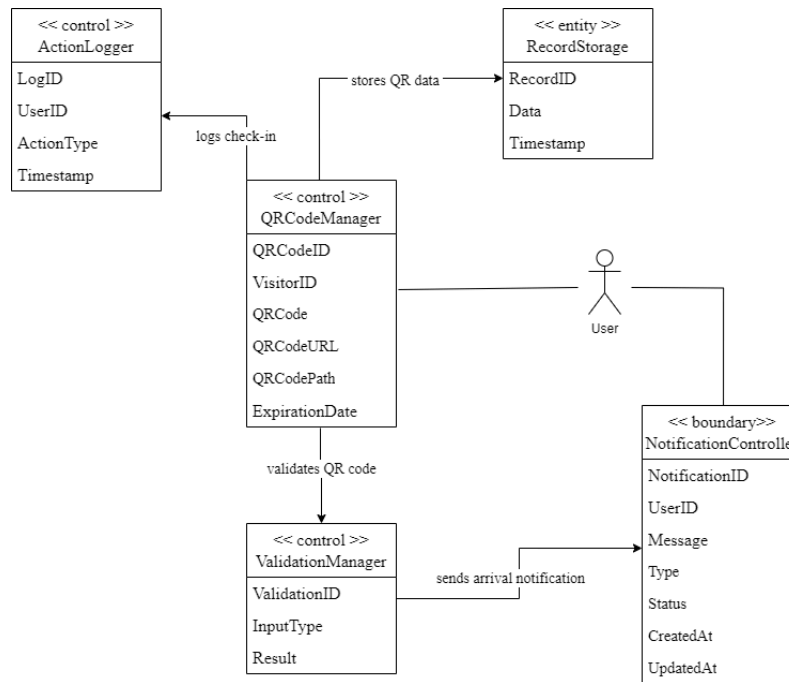


Figure 3. Domain Model for ScanQR(UC-7)

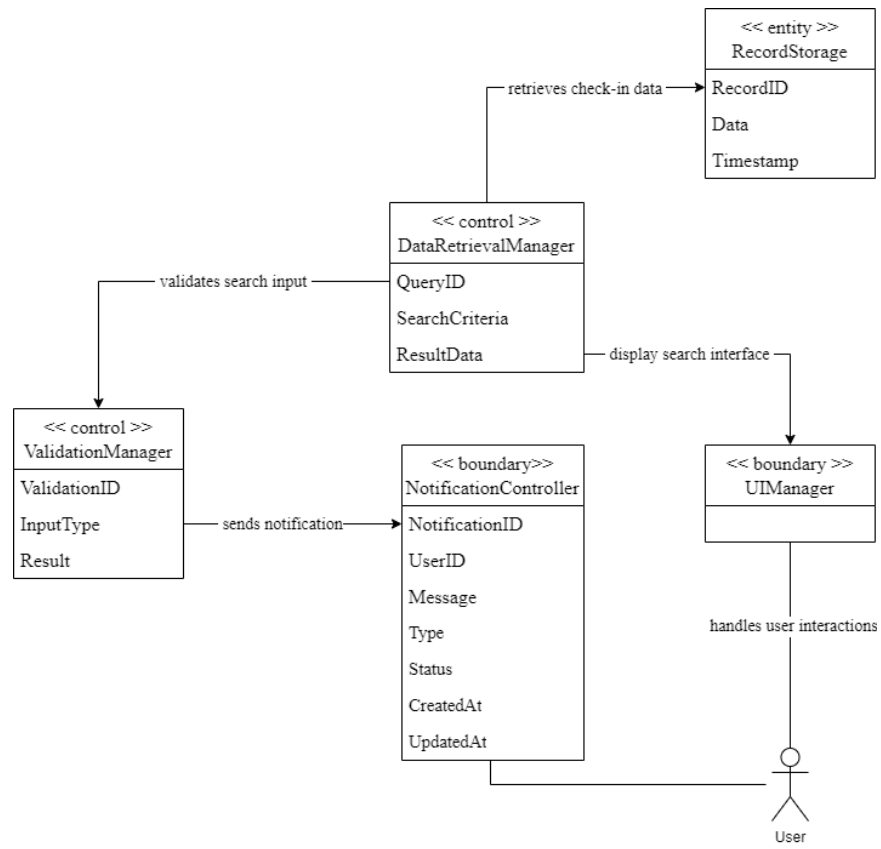


Figure 4. Domain Model for ViewHistory (UC-8)

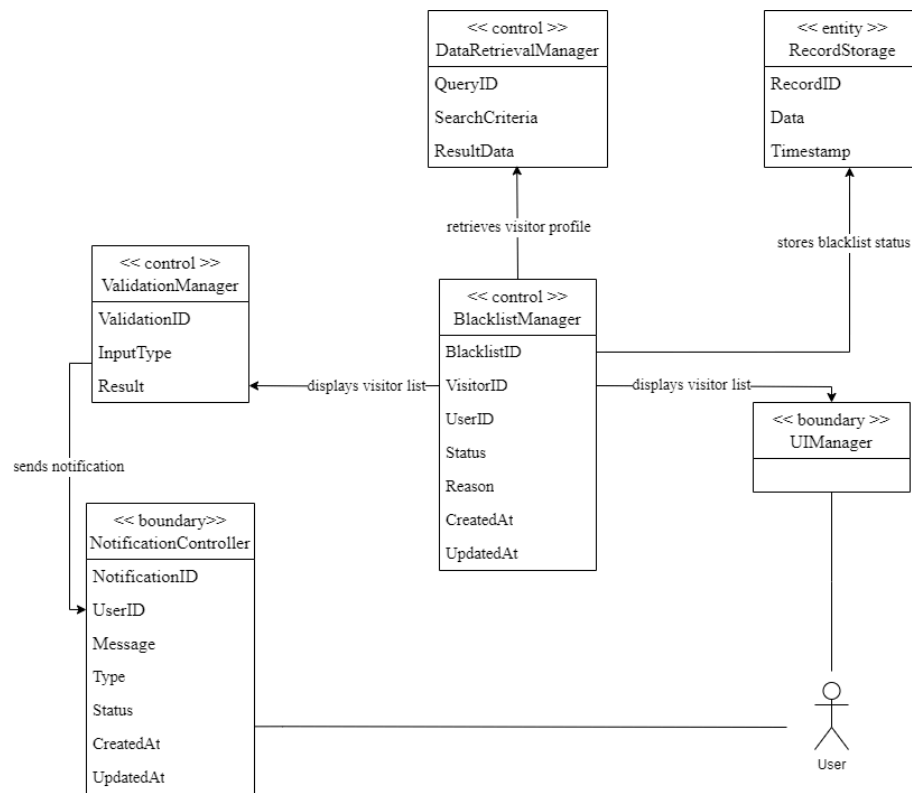


Figure 5. Domain Model for ManageBlacklist(UC-9)

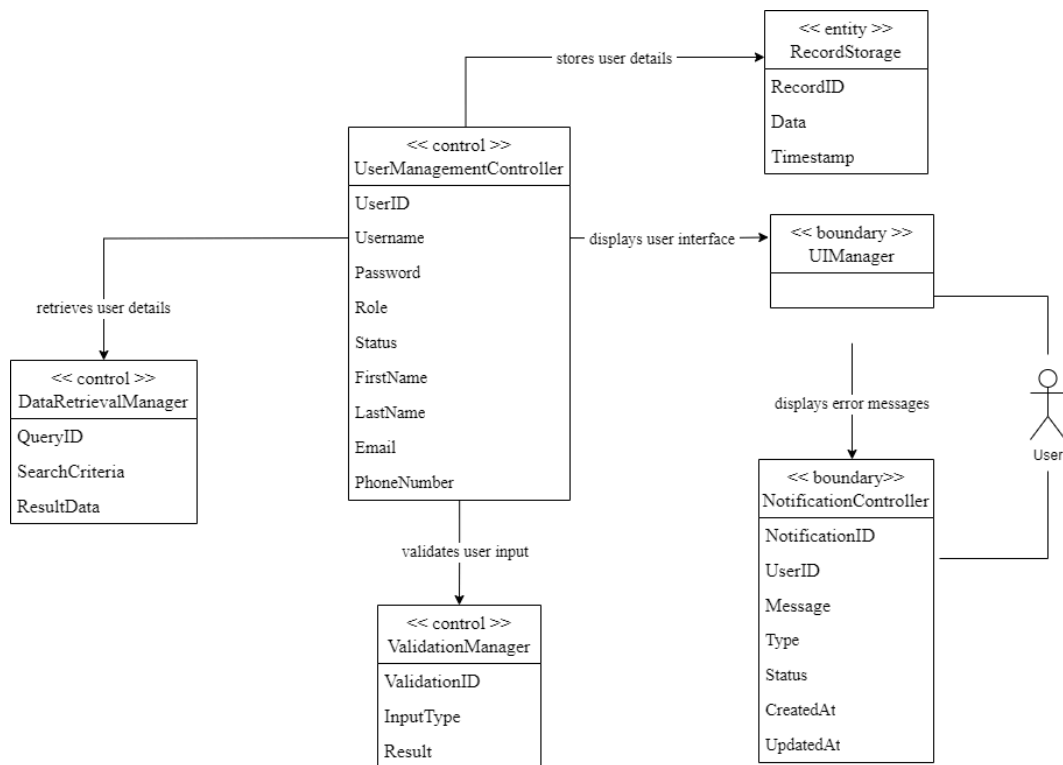


Figure 6. Domain Model for ManageUsers (UC-12)

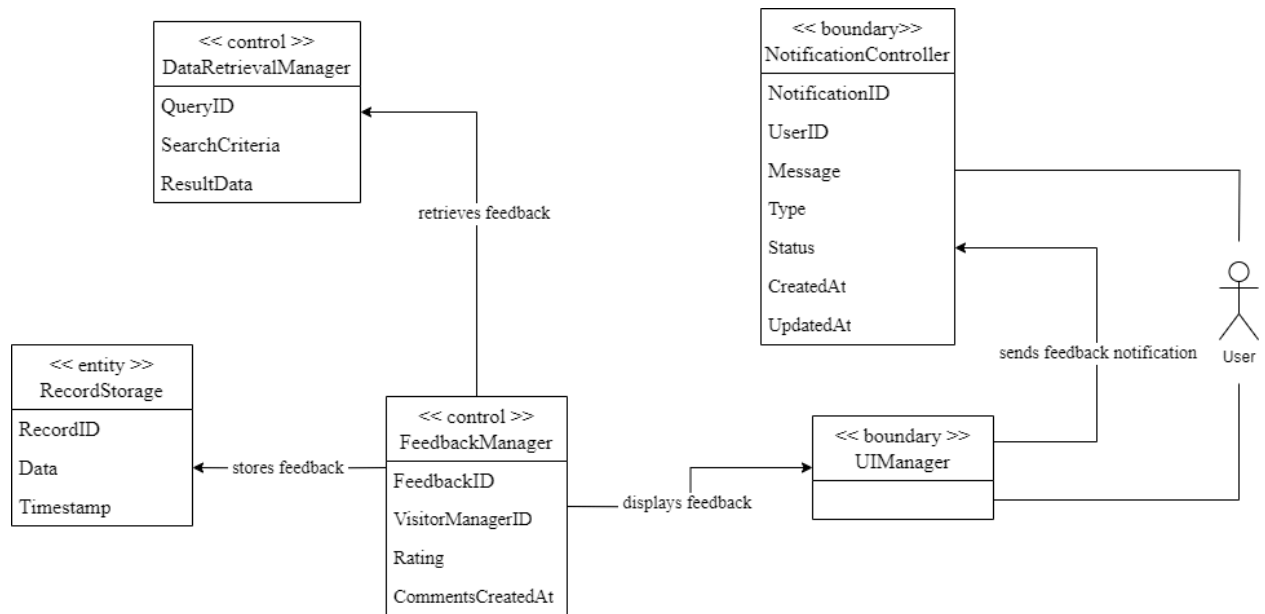


Figure 7. Domain Model for ViewFeedback (UC-16)

System Operation Contracts

Contract Name	CreateVisitSchedule
Operation	registerVisitor(visitorDetails, schedule)
Actor:	Resident
Cross Reference	UC-2 (CreateVisitorSchedule)
Responsibilities	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • Resident is authenticated (NONREQ-8 enforced). • Visitor details (name, contact, frequency, duration) comply with input validation rules (REQ-16, REQ-17).
Post Conditions	<ul style="list-style-type: none"> • 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 17. System Contracts for CreateVisitSchedule (UC-2)

Contract Name	SubmitFeedback
Operation	submitFeedback(visitorID, rating, comments)
Actor	Visitor
Cross Reference	UC-6 (SubmitFeedback)
Responsibilities	<ul style="list-style-type: none"> • Enforce feedback form validation (e.g., rating 1–5, max 500 characters). • Anonymize feedback for compliance with NONREQ-16.

Preconditions	<ul style="list-style-type: none"> • Visitor check-in data exists (REQ-3). • Feedback link accessed via valid session (NONREQ-15).
Post Conditions	<ul style="list-style-type: none"> • 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 18. System Contracts for SubmitFeedback (UC-6)

Contract Name	ScanQR
Operation	scanQR(qrCode)
Actor	Security Guard
Cross Reference	UC-7 (ScanQR)
Responsibilities	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • QR code format is valid (REQ-12). • Security guard has system access (NONREQ-6).
Post Conditions	<ul style="list-style-type: none"> • 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 • A Notification object is created and sent to the associated Resident (UC-5) • 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 19. System Contracts for ScanQR(UC-7)

Contract Name	ViewHistory
Operation	viewVisitorHistory(filters)
Actor	Security Guard

Cross Reference	UC-8 (ViewHistory)
Responsibilities	<p>Process filter criteria (date ranges, visitor names) from input validation rules (NONREQ-3)</p> <p>Retrieve records matching:</p> <ul style="list-style-type: none"> • Visitor ID/name • Check-in timestamps • Access status (granted/denied) <p>Handle empty results by displaying:</p> <ul style="list-style-type: none"> • "No records found" for valid queries <p>"Database connection error" for system failures</p>
Preconditions	<p>Security guard is authenticated (NONREQ-8 enforced)</p> <p>Visitor check-in data exists in the database (REQ-11)</p>
Post Conditions	<ul style="list-style-type: none"> • 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 20. System Contracts for ViewHistory (UC-8)

Contract Name	ManageBlacklist
Operation	fetchBlacklist(visitorID, action)
Actor	Resident/Security Guard
Cross Reference	UC-9 (ManageBlacklist)
Responsibilities	<ul style="list-style-type: none"> • Restrict blacklist modifications to authorized roles. • Propagate blacklist updates to check-in validation workflows.
Preconditions	<ul style="list-style-type: none"> • Requester has blacklist management privileges (REQ-15).

	<ul style="list-style-type: none"> • Visitor exists in the database (REQ-4).
Post Conditions	<ul style="list-style-type: none"> • 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 21. System Contracts for ManageBlacklist(UC-9)

Contract Name	ManageUsers
Operation	modifyUser(userID, action, details)
Actor	Administrator
Cross Reference	UC-12 (ManageUsers)
Responsibilities	<ul style="list-style-type: none"> • Encrypt sensitive data (e.g., passwords) during storage (NONREQ-19). • Synchronize user roles with permission matrices.
Preconditions	<ul style="list-style-type: none"> • Administrator has elevated permissions (REQ-4). • Required fields (e.g., email, role) provided (NONREQ-18).
Post Conditions	<ul style="list-style-type: none"> • 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 22. System Contracts for ManageUsers (UC-12)

Contract Name ViewFeedback	
Operation	fetchFeedback(filters)
Actor	Admin/Resident/Security
Cross Reference	Use cases: ViewFeedback
Responsibilities	<ul style="list-style-type: none"> • Apply role-based data filtering (e.g., residents see only their visitors). • Handle empty result sets with user-friendly prompts.
Preconditions	<ul style="list-style-type: none"> • User has feedback-viewing permissions (REQ-3). • Filters (date, visitor type) are valid (NONREQ-17).
Post Conditions	<ul style="list-style-type: none"> • 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 23. 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. Resident Vehicles: Maintains a record of registered resident vehicles.

Attributes: id (Primary Key) – Unique vehicle record ID, user_id – Associated resident (Foreign Key → users.id), vehicle_make, vehicle_model, vehicle_color – Car details, license_plate – Vehicle registration number

6. 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

7. Entry Logs: Records all visitor and resident scans for security tracking.

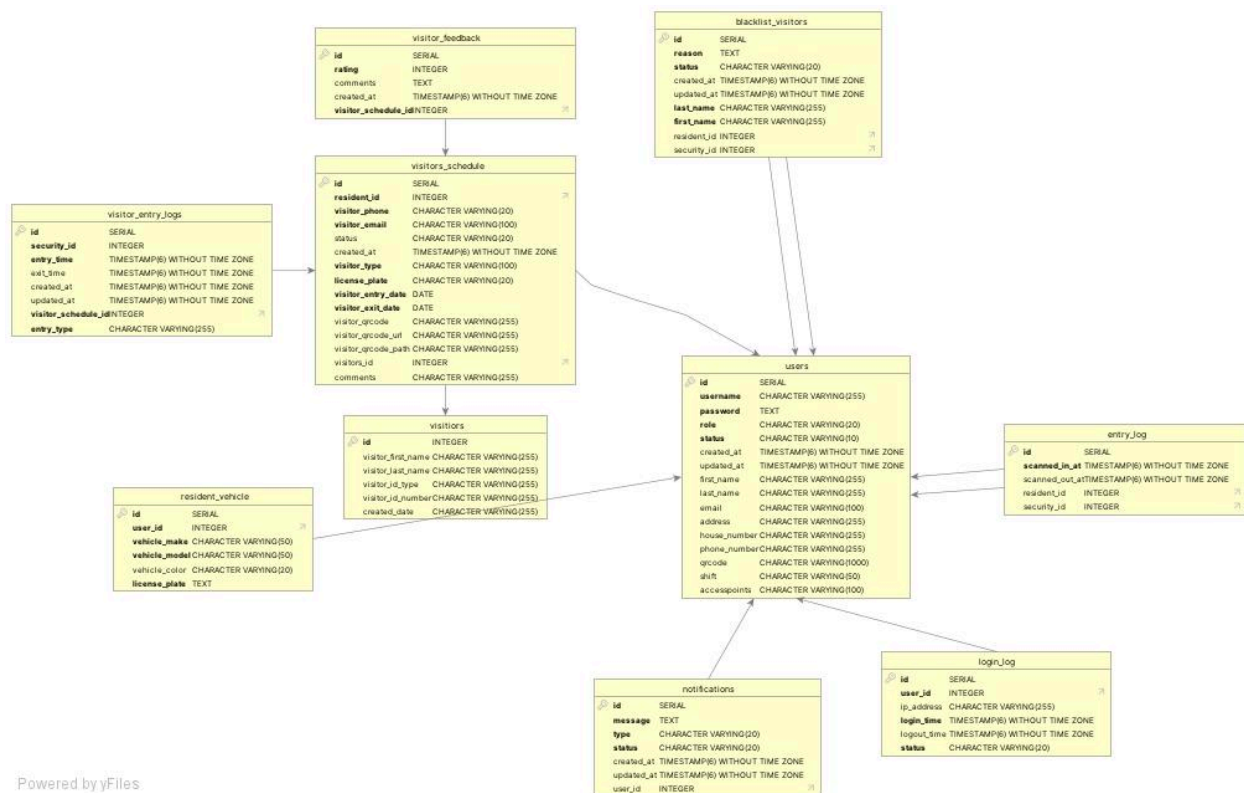
Attributes: id (Primary Key) – Unique log ID, user_id – Associated resident or visitor (Foreign Key → users.id or visitors.id), scanned_in_at, scanned_out_at – Timestamps for tracking entry/exit, security_id – Security personnel ID who scanned the visitor (Foreign Key → users.id)

8. 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

9. 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



Powered by yFiles

Figure 8. 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

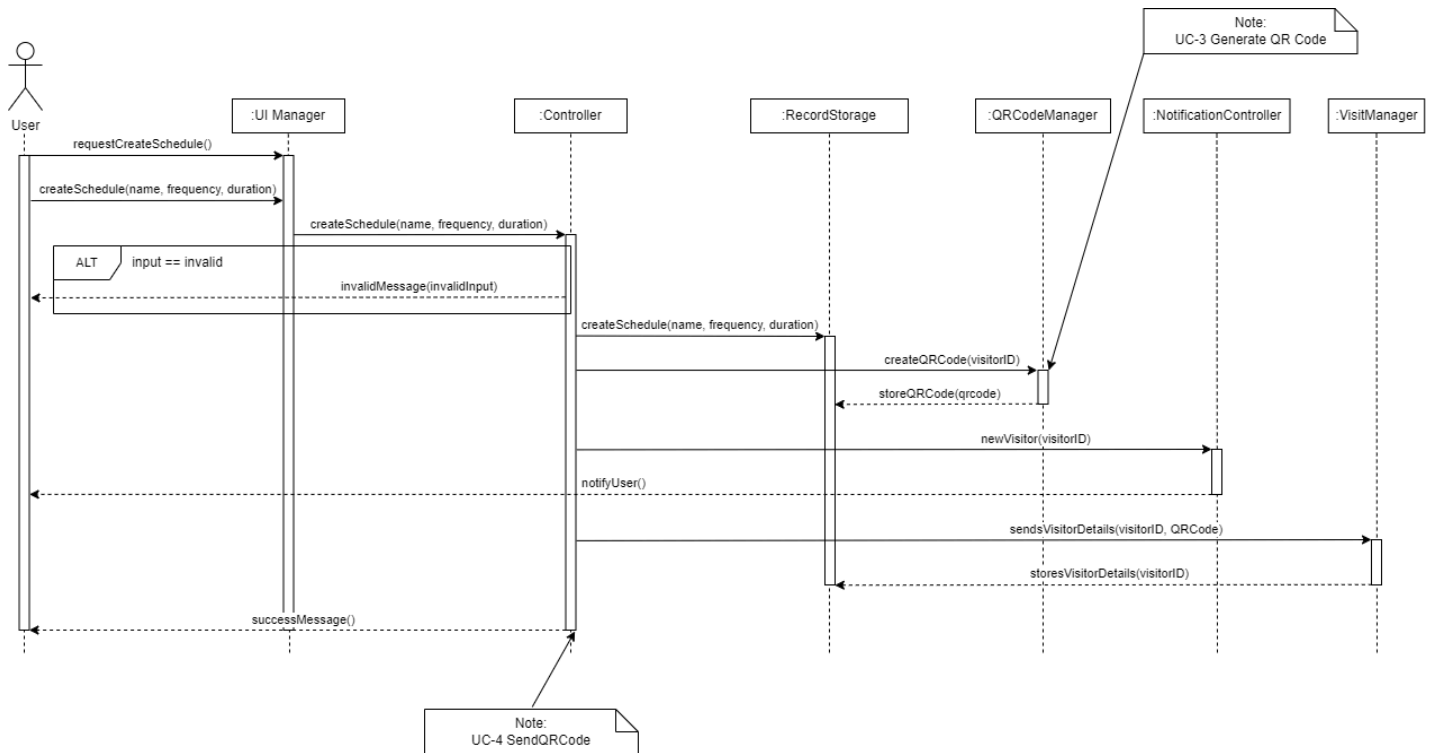


Figure 9. CreateVisitSchedule (UC-2)

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.

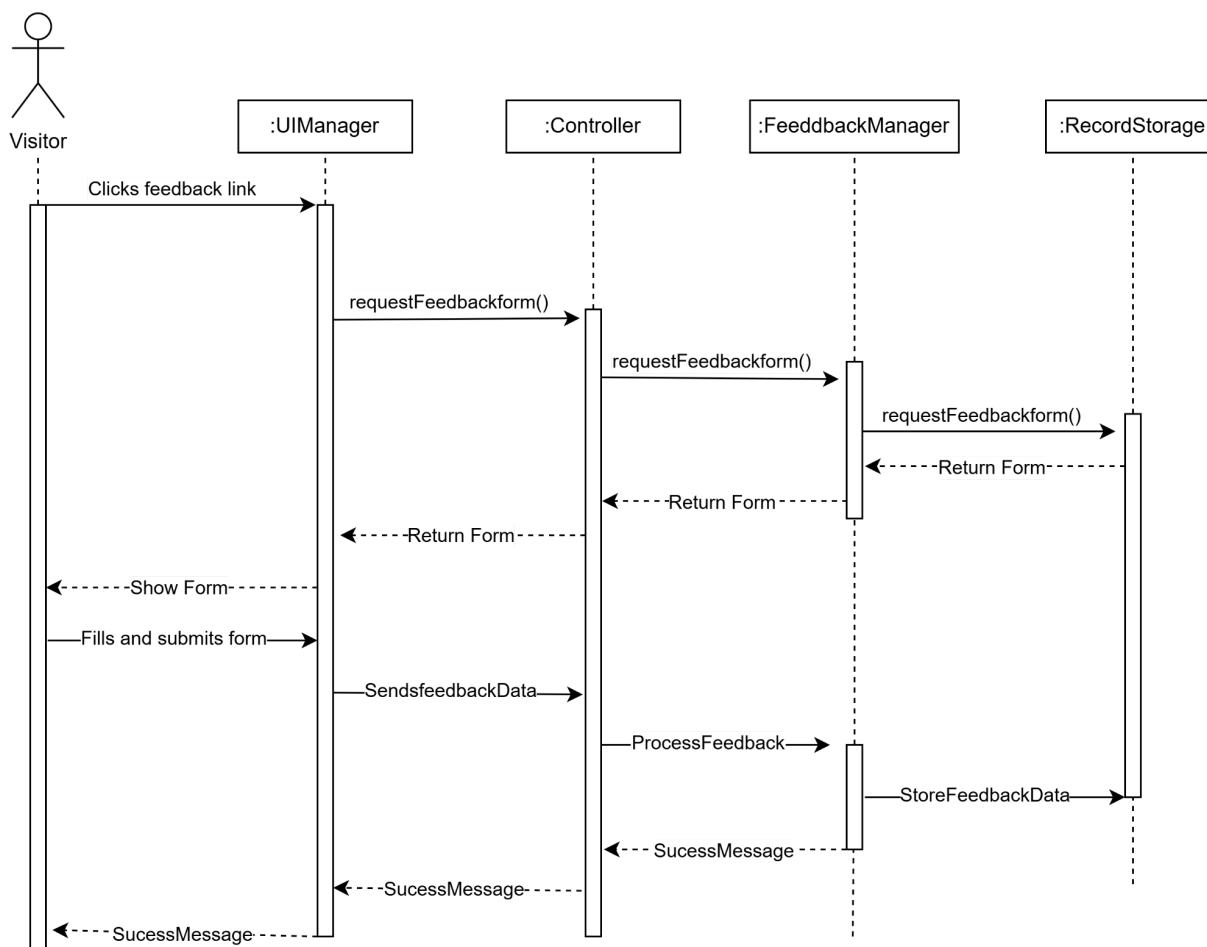


Figure 10. SubmitFeedback (UC-6)

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.

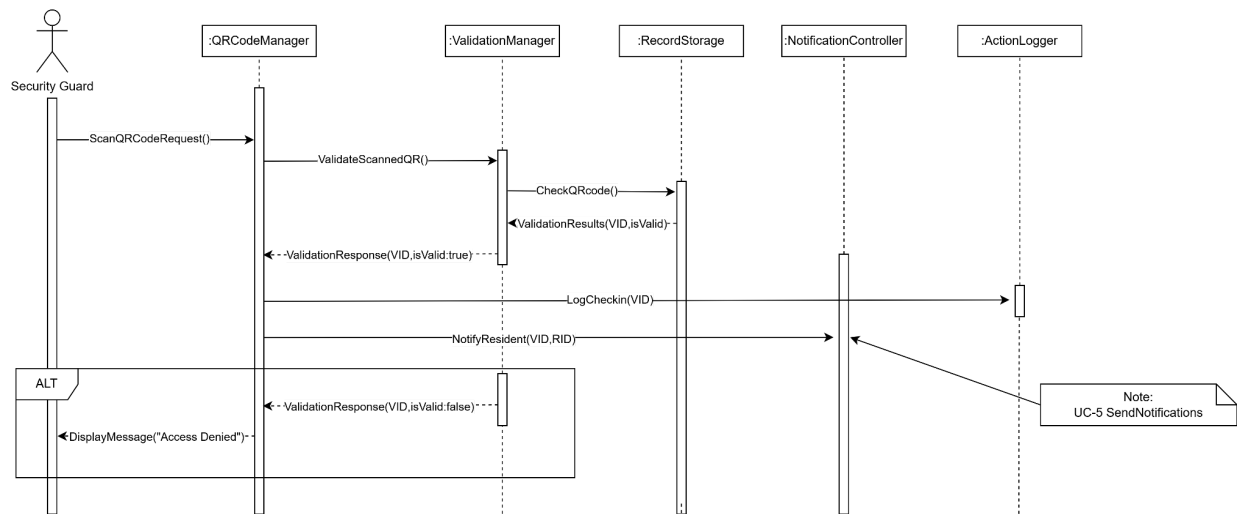


Figure 11. ScanQR(UC-7)

The Security Guard initiates a **ScanQRCodeRequest()** to the **QRCodeManager**. The **QRCodeManager** then calls **ValidateScannedQR()** on the **ValidationManager** to verify the scanned QR code. The **ValidationManager** subsequently requests a **CheckQRCode()** operation from the **RecordStorage** to cross-check the QR code with stored visitor data.

Once the **RecordStorage** processes the request, it returns a **ValidationResult(VID, isValid)** to the **ValidationManager**, indicating whether the scanned QR code is valid. The **ValidationManager** then sends a **ValidationResponse(VID, isValid: true)** back to the **QRCodeManager** if the QR code is valid. Upon successful validation, the **QRCodeManager** logs the check-in attempt by invoking **LogCheckin(VID)** on the **ActionLogger**. Following this, the **QRCodeManager** sends a **NotifyResident(VID, RID)** request to the **NotificationController**, which triggers a notification to inform the respective resident about the visitor's arrival.

If the QR code is invalid, the **ValidationManager** instead returns a **ValidationResponse(VID, isValid: false)** to the **QRCodeManager**. Consequently, the **QRCodeManager** triggers an alternative flow where it calls **DisplayMessage("Access Denied")**, notifying the security guard that the QR code is invalid, preventing unauthorized entry.

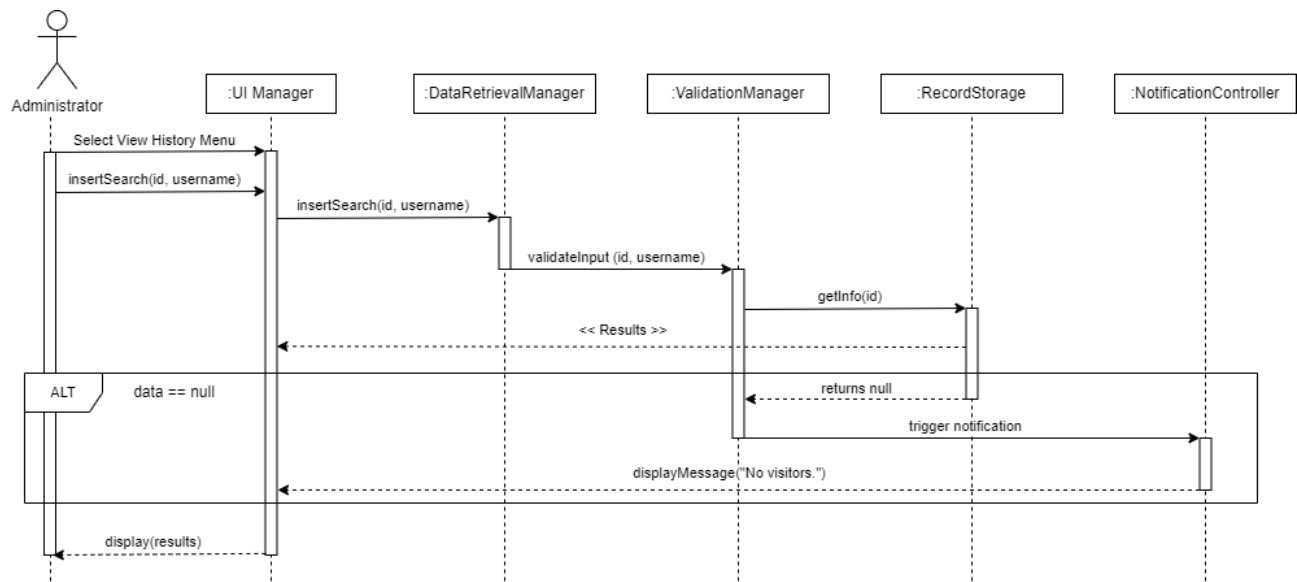


Figure 12.ViewHistory (UC-8)

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.

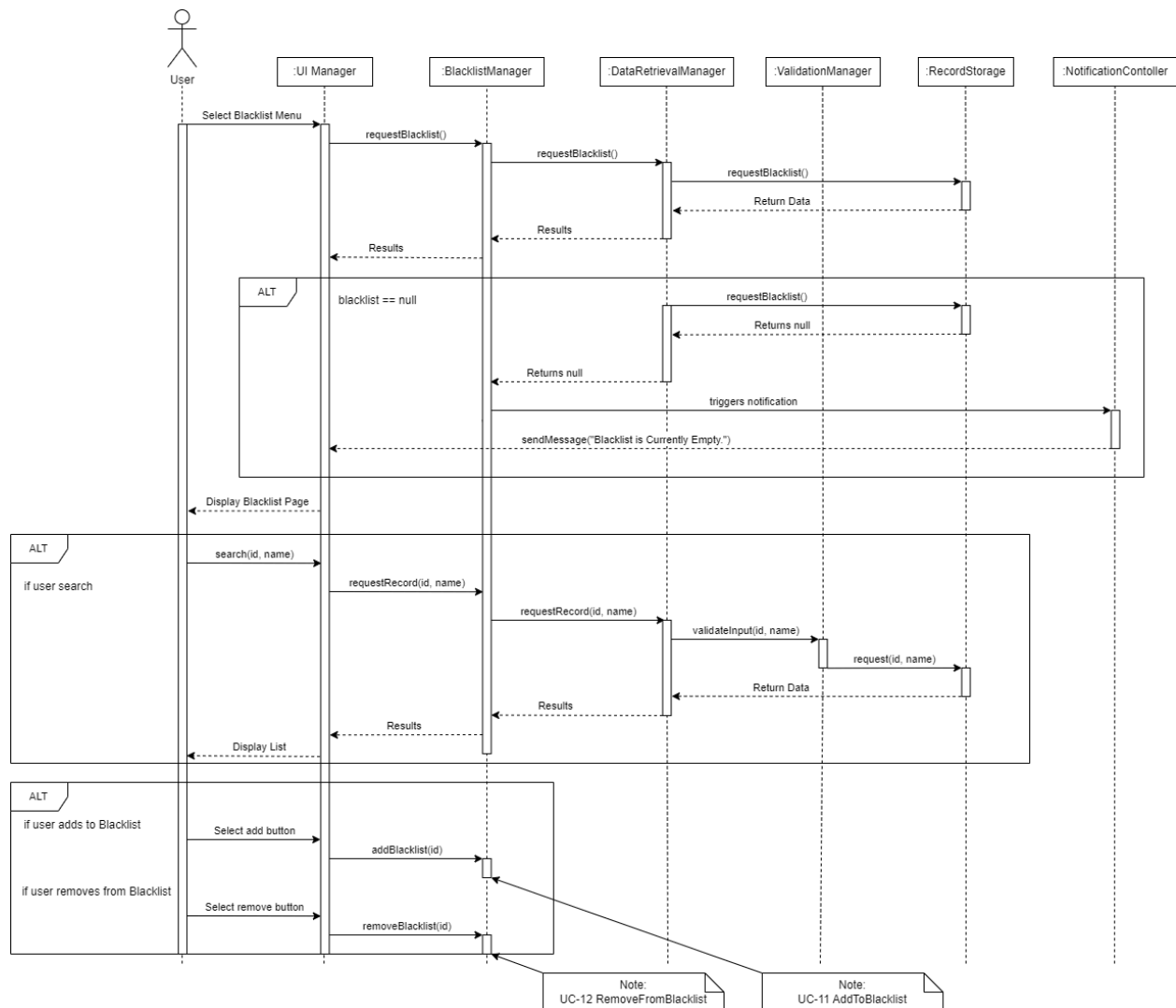


Figure 13. ManageBlacklist(UC-9)

When a user wants to manage the blacklist in the gated community system, they begin by selecting the **Blacklist Menu** in the UI. This action prompts the **UI Manager** to send a requestBlacklist() to the **BlacklistManager**, which in turn requests blacklist data from the **DataRetrievalManager**. The **DataRetrievalManager** forwards the request to **RecordStorage**, where the system retrieves the blacklist records. The results are then passed back through the chain to the **UI Manager**, which displays the blacklist page to the user.

If the blacklist is empty, an alternate flow (ALT) is triggered. In this case, **RecordStorage** returns null, indicating that no blacklisted records exist. The **BlacklistManager** triggers a notification via the **NotificationController**, and the system sends a message to the user stating, "Blacklist is Currently Empty."

Once the blacklist page is displayed, the user has several interaction options. If they choose to search for a specific record, they enter an ID and name, which triggers the **UI Manager** to send a search(id, name) request. This request is processed by the **BlacklistManager**, which retrieves the record from the **DataRetrievalManager**. The **ValidationManager** verifies the input before requesting the corresponding record from **RecordStorage**. The results are then returned and displayed to the user.

Additionally, the user can modify the blacklist by either adding or removing an individual. If the user selects the "Add" button, the **UI Manager** sends an addBlacklist(id) request, which follows the **UC-11 AddToBlacklist** process. If the user selects the "Remove" button, a removeBlacklist(id) request is sent, following the **UC-12 RemoveFromBlacklist** process. These actions ensure that the blacklist remains updated according to the user's preferences.

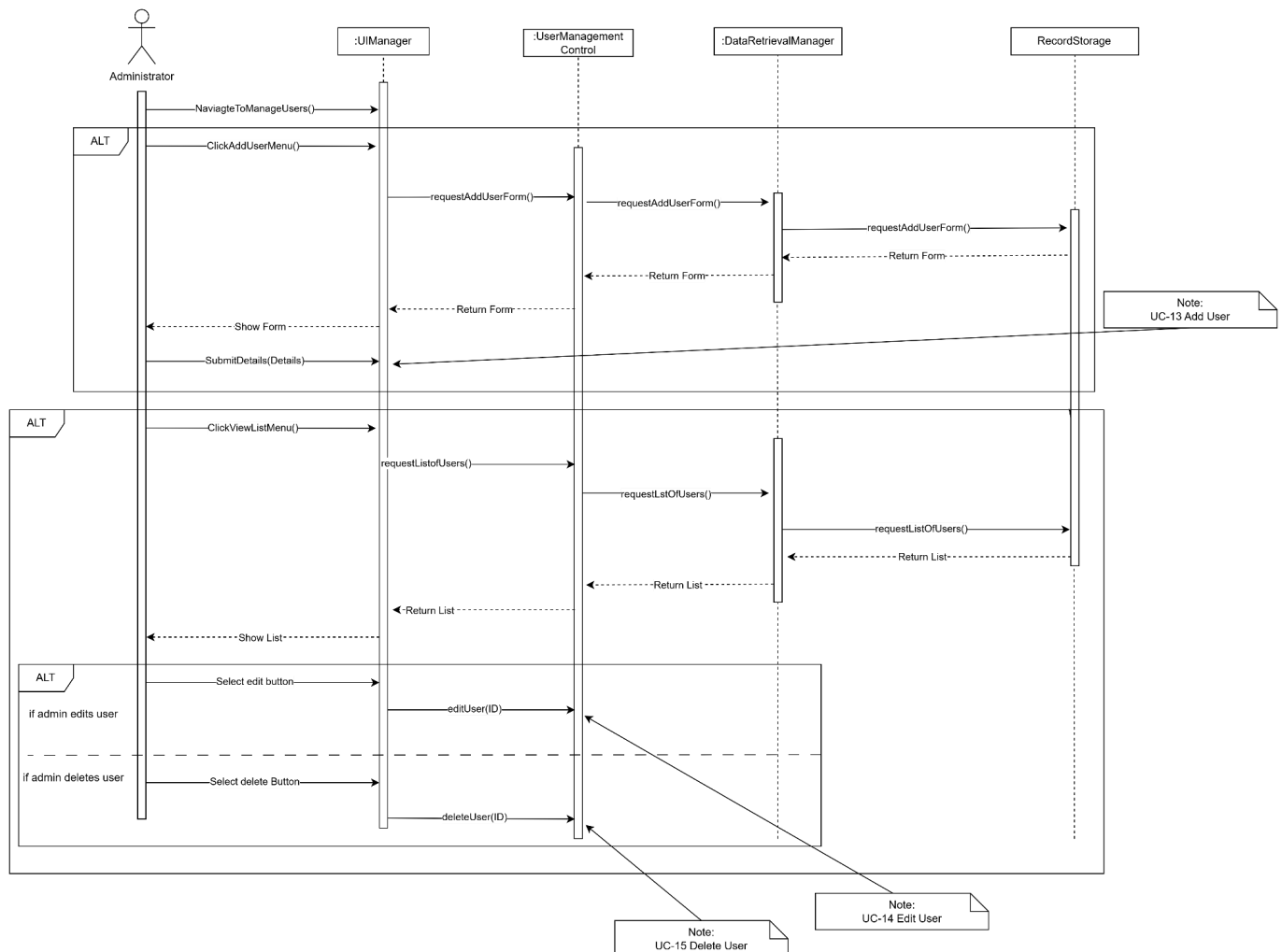


Figure 14. ManageUsers (UC-12)

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.

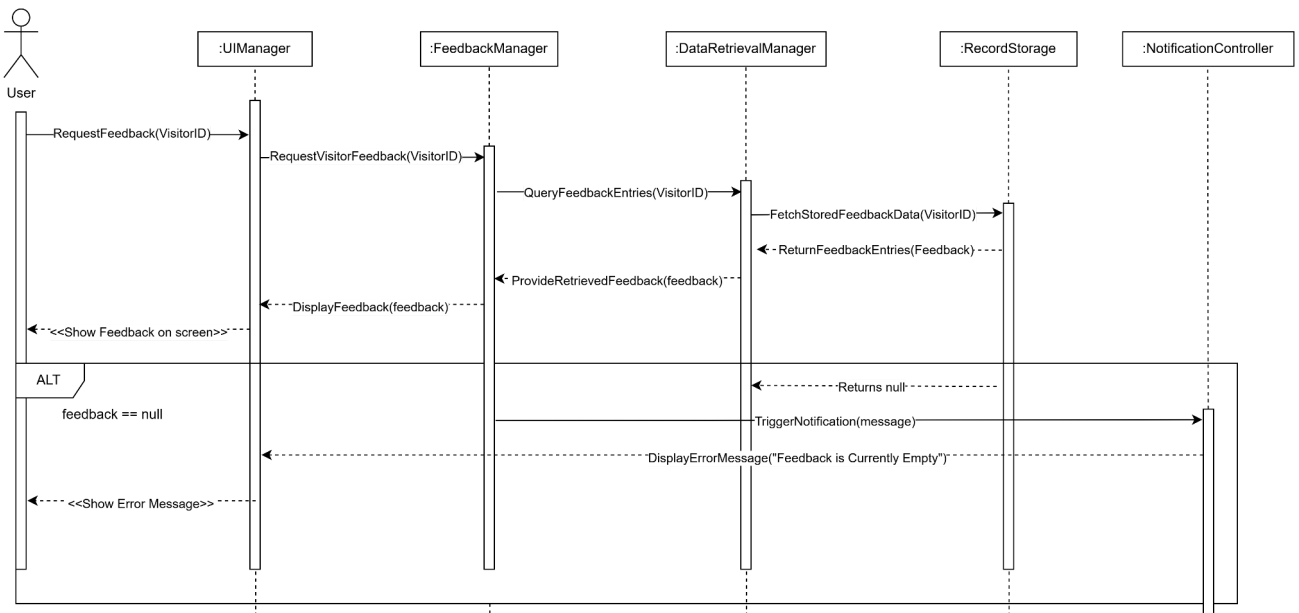


Figure 15. ViewFeedback (UC-16)

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.

Class Diagram and Interface Specification

Class Diagram

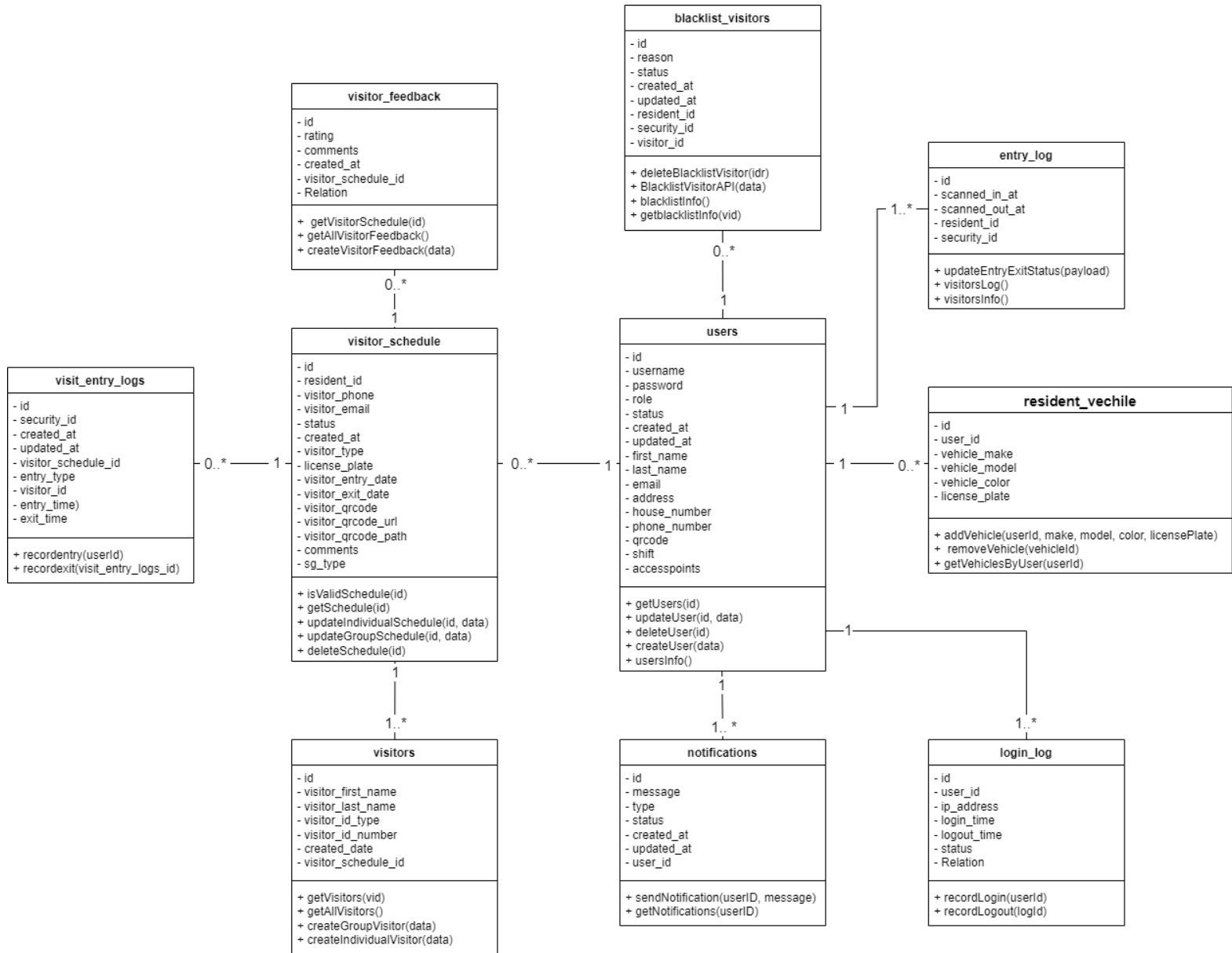


Figure 16. Class Diagram

Data Types and Operation Signatures

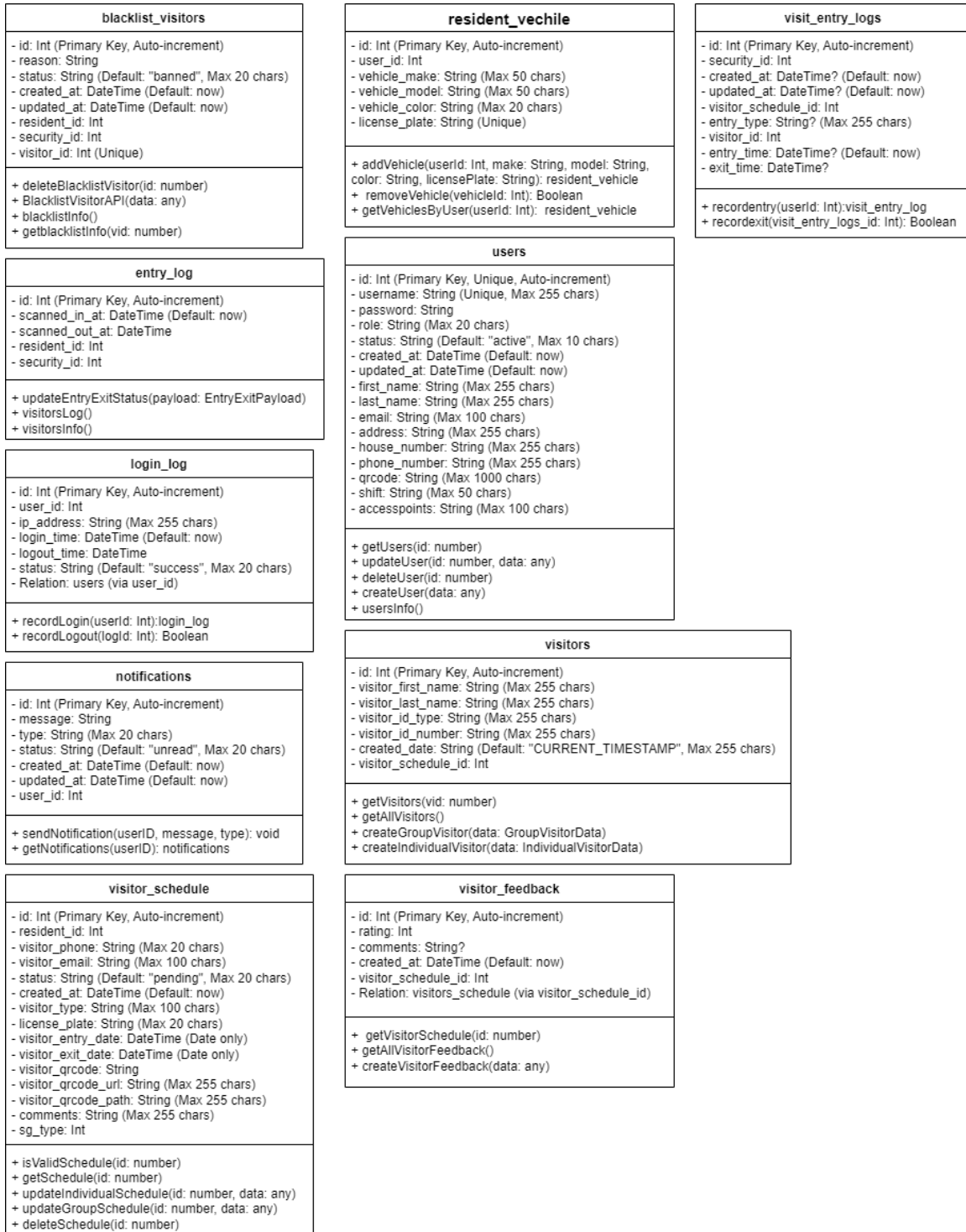


Figure 17. Data Types and Operation Diagram

Traceability Matrix

	Login Log	Entry Log	Notification	Users	Blacklist Visitors	Visitor Feedback	Visitors	Visitor Schedule	Visitor Entry Log	Resident Vehicle
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 24. 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	<p>Input Validation: Check if the user is using a QR code or manual entry request.</p> <p>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).</p> <p>For manual entry: Verify user details against the database (e.g., VisitorID, Name, Contact Details).</p> <p>Blacklist Check: Query the database to confirm if the user/visitor is blacklisted.</p> <p>If blacklisted: Deny entry and log the attempt with details (time, reason for denial) and notify security personnel via the dashboard.</p> <p>Access Authorization: If the user passes validation: Update visitor status to "Active" in the database. Log entry details (VisitorID, EntryTime, EntryType).</p>
3	Real-Time Dashboard Update	<p>Active Attendees:</p> <ul style="list-style-type: none"> - 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. <p>Blacklisted Attendees:</p> <ul style="list-style-type: none"> - 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	<ul style="list-style-type: none"> - 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	<ul style="list-style-type: none"> - Allow administrators to view detailed reports via the dashboard: - Daily active attendee counts. - Entry attempts and blacklist updates.

Table 25. 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

Visitor First Name <input type="text" value="Visitor First Name"/> <small>Enter Visitor First Name</small>	Visitor Last Name <input type="text" value="Last Name"/> <small>Enter Visitor Last Name</small>
Visitor Phone Number <input type="text" value="Visitor Phone Number"/> <small>Enter Visitor Phone Number</small>	Visitor Email Address <input type="text" value="Email Address"/> <small>Enter Visitor Email Address</small>
Visitor ID type <input style="border: 1px solid #ccc; background-color: #f9f9f9;" type="text" value="ID Type"/> <small>Select Visitor ID Type</small>	Visitor ID Number <input type="text" value="ID Number"/> <small>Enter Visitor ID Number</small>
Visitor License Plate <input type="text" value="Visitor License Plate"/> <small>Enter Visitor License Plate</small>	<input type="checkbox"/> Recurring Visitor <small>Recurring</small>
Entry Date Time <input type="text" value="04/11/2025 09:15 AM"/> <small>Enter Visit Day and Time</small>	Exit Date Time <input type="text" value="04/11/2025 09:15 AM"/> <small>Enter Exit Date Time</small>
Comments <input type="text" value="Comments"/> <small>Enter Comments</small>	Status <input style="border: 1px solid #ccc; background-color: #f9f9f9;" type="text" value="Status"/> <small>Status of Visitor</small>

Figure 18. 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.

Visitor Phone Number <input type="text" value="Visitor Phone Number"/> <small>Enter Visitor Phone Number</small>	Visitor Email Address <input type="text" value="Email Address"/> <small>Enter Visitor Email Address</small>
Visitor License Plate <input type="text" value="Visitor License Plate"/> <small>Enter Visitor License Plate</small>	<input type="checkbox"/> Recurring Visitor <small>Recurring</small>
Entry Date Time <input type="text" value="04/11/2025 09:16 AM"/> <small>Enter Visit Day and Time</small>	Exit Date Time <input type="text" value="04/11/2025 09:16 AM"/> <small>Enter Exit Date Time</small>
Comments <input type="text" value="Comments"/> <small>Enter Comments</small>	Status <input type="text" value="Status"/> <small>Status of Visitor</small>

Figure 19. 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

List Visitors

Filter First Name...						
Visitor First Name	Visitor Last Name	License Plate	Visitor Type	Single or Group	Status	Actions
kelsey	aban	123 kda	recurring	single visitor	inactive	...
aiyesha	coleman	354 dey	one-time	single visitor	inactive	<div> Actions </div> <div>View Schedule</div>

Figure 20. 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

Visitor Details

Phone: 606-0222	Email: kelsey@gmail.com	License Plate: 123 KDA
Visitor Type: recurring	Status: inactive	

Scheduled Entry: Thu Apr 10 2025
Scheduled Exit: Thu Apr 10 2025
Comments: visiting a friend

QRCode




Figure 21. 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**Existing Visitors**

Filter First Name...				
Visitor First Name	Visitor Last Name	ID Type	ID Number	Actions
alyesha	coleman	passport	2	...

Actions
Add to Blacklist

Figure 22. 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."

Blacklist Visitor

First Name Alyesha	Last Name Coleman
ID Type passport	ID Number 2
Status Banned	

Reason
Reason for blacklisting

Provide a reason for blacklisting this visitor.

Submit
Cancel

Figure 23. 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.

Blacklist Visitors

Filter First Name...

Visitor First Name	Visitor Last Name	Visitor ID Type	Visitor ID Number	Status	Actions
kelsey	aban	ss	1	banned	...

Actions
 Update Blacklist Reason
 Delete Blacklist Visitor

Figure 24. 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.		Success (200)
3. Call function createVisitSchedule(userId, input) with valid user ID.		Success (200)
4. Call function notifyUser(userId)		Success (200)

Table 26. 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.		Success (200)
3. Call function validateFeedback(input) with empty feedback.		Failure (400)
4. Call function validateFeedback(input) with invalid rating (e.g. 6).		Failure (400)
5. Call function submitFeedback(userId, feedback) with valid user ID and feedback.		Success (200)
6. Call function submitFeedback(userId, feedback) with non-existent user ID.		Success (404)

Table 27. 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 28. 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 29. 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 30. 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 31. 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.		Success
2. Call function validateUserInput(input) with valid name, email, and role.		Success (200)
3. Call function validateUserInput(input) with empty name.		Failure (400)
4. Call function checkDuplicateUser(email) with the same email.		Success (200)
5. Call function checkDuplicateUser(email) with existing email.		Failure (400)
6. Call function createUser(input) with new user data.		Failure (400)
7. Call function createUser(input) with invalid role.		Success (200)

Table 32. 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		Success
2. Call function getUserById(userId)		Success (200)
3. Call function updateUser(userId, newData)		Success (200)

Table 33. 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:
<ol style="list-style-type: none"> 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 34. 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:
<ol style="list-style-type: none"> 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 35. 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.

Responsibilities	Team Members				
	Aiyesha Coleman	Kelsey Aban	Jordani Alpuche	Aiden Pinelo	Arthur Butler
Part 1					
Analysis and Domain Modeling	✓	✓	✓	✓	-
Part 2					
Interaction Diagram	✓	✓	✓	-	-
Class Diagram and Interface Specification	✓	✓	✓	-	-
Part 3					
Algorithms and Data Structure	✓	✓	✓		✓
User Interface and Implementation	✓	✓	✓		
Project Management and Plan of Work	✓	✓	✓		

Table 36. Breakdown of Work Contribution

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.

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

Tasks in Progress:

- 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, the team is actively working on the final phases of development, including ongoing debugging and refining the UI. Report #3 and Demo #2 are currently underway, with expected completion by early May. The Application Interface development and the E-Archive component are still in progress, extending into late May.

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

Future development:

- 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.

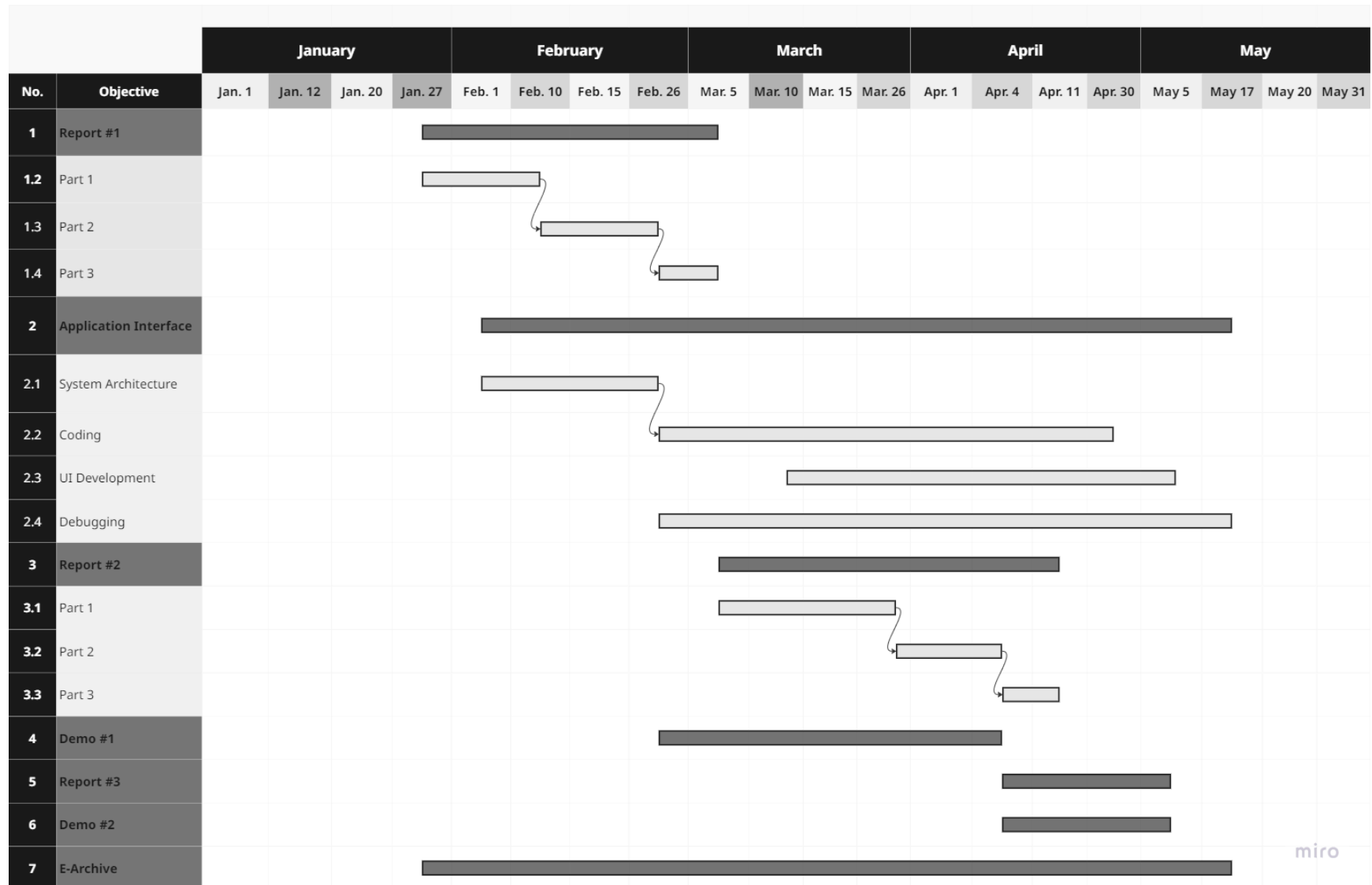


Figure 25. Gantt Chart

References

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>