Report #1 — Domain Modeling & CRUD Validation

I. Domain Classes: Noun Technique

This section identifies and models the domain classes using the *Noun Technique*.

Step 1 — Extract Nouns

Nouns were extracted from:

- User stories (Deliverable #2)
- Use case descriptions
- System events
- Domain knowledge
- Provided UML class diagram **Table 1 — Initial Raw Noun List**

(All nouns found before filtering)

Student, Faculty, Academic Affairs, User, QR Code, Fingerprint, Attendance, Session, Course, Enrollment, Dashboard, Location, Time, Record, Status, Device, Biometric Scanner, Attendance Summary, Warning Notification, Report, Threshold, Semester, Integration Service, FMS, Major, Email, Role.

Step 2 — Classify Nouns (Include / Exclude / Research)

Noun	Decision Reason
Student	**Include System stores student data**
Faculty	**Include Faculty manage sessions**
Academic Affairs	**Include Responsible for reports**
User	**Include Generalization superclass**
Course	**Include Sessions belong to courses**
Enrollment	**Include Student–course association**
Noun	**Decision Reason**
Attendance Session	**Include Core unit of attendance**

Attendance Record	**Include Tracks student status**
Biometric Device	**Include Required for fingerprint method**
QR Code	**Include Method of taking attendance**
Integration Service	**Include Pushes data to FMS**
FMS System	**Include External system integrated**
Report	**Include Academic Affairs reports**
Status	**Exclude Attribute**
Location	**Exclude Attribute**
Time	**Exclude Attribute**
Fingerprint	**Exclude Attribute/temporary input**
Dashboard	**Exclude UI element**
Warning Notification	**Exclude System output only**

Step 3 — Final Master List of Domain Classes

Final Approved Domain Classes:

1. User
2. Student
3. Faculty
4. AcademicAffairs
5. Course
6. Enrollment
7. AttendanceSession
8. AttendanceRecord
9. BiometricDevice
10. QRCode
11. IntegrationService

12. FMSSystem
13. Report

II. Use Cases and CRUD Technique

CRUD helps ensure that use cases from Deliverable #2 are complete, correct, and fully support all domain classes.

II.a CRUD Matrix (Step-by-Step)

Domain Class	Create	Read	Update	Delete
Student	Enrollment	View Sessions	Correct Attendance	—
Faculty	Admin Add	View Records	Make Corrections	—
AcademicAffairs	Admin Add	Generate Reports	—	—
Course	Create Course	View Course	Update Course	Delete Course
Enrollment	Add Student	View Enrollment	Update Enrollment	Remove Enrollment
AttendanceSession	Create Session	View Dashboard	Close Session	—
AttendanceRecord	QR/Fingerprint	View Attendance	Manual Correction	—
BiometricDevice	Register Device	Validate	Update Device	Delete Device
QRCode	Generate	Validate	—	Auto-expire

Report	Generate	View	Regenerate	Delete
IntegrationService	—	Monitor Sync	Push to FMS	—
FMS System	Register	Receive Data	—	—

II.b CRUD Findings & Required Use Case Refinements

Based on CRUD analysis:

New Use Cases Needed

1. **Manage Enrollment** ○ Supports Create/Update/Delete for Enrollment class.
2. **Register Biometric Device** ○ Required for device creation and updates.
3. **View Attendance History** ○ CRUD shows “Read” must allow viewing past attendance.

Existing Use Cases Confirmed

- Record Attendance via QR
- Record Attendance via Fingerprint
- Create Attendance Session
- View Real-time Attendance
- Correct Attendance Record
- Generate At-Risk Student Report
- Generate End-of-Term Report
- Push Data to FMS

II.c Updated Use Case Diagram (Textual Description)

Actors:

- Student
- Faculty

- Academic Affairs
- FMS System

Use Cases:

Student

- Record Attendance via QR
- Record Attendance via Fingerprint
- View Attendance History *(New)*

Faculty

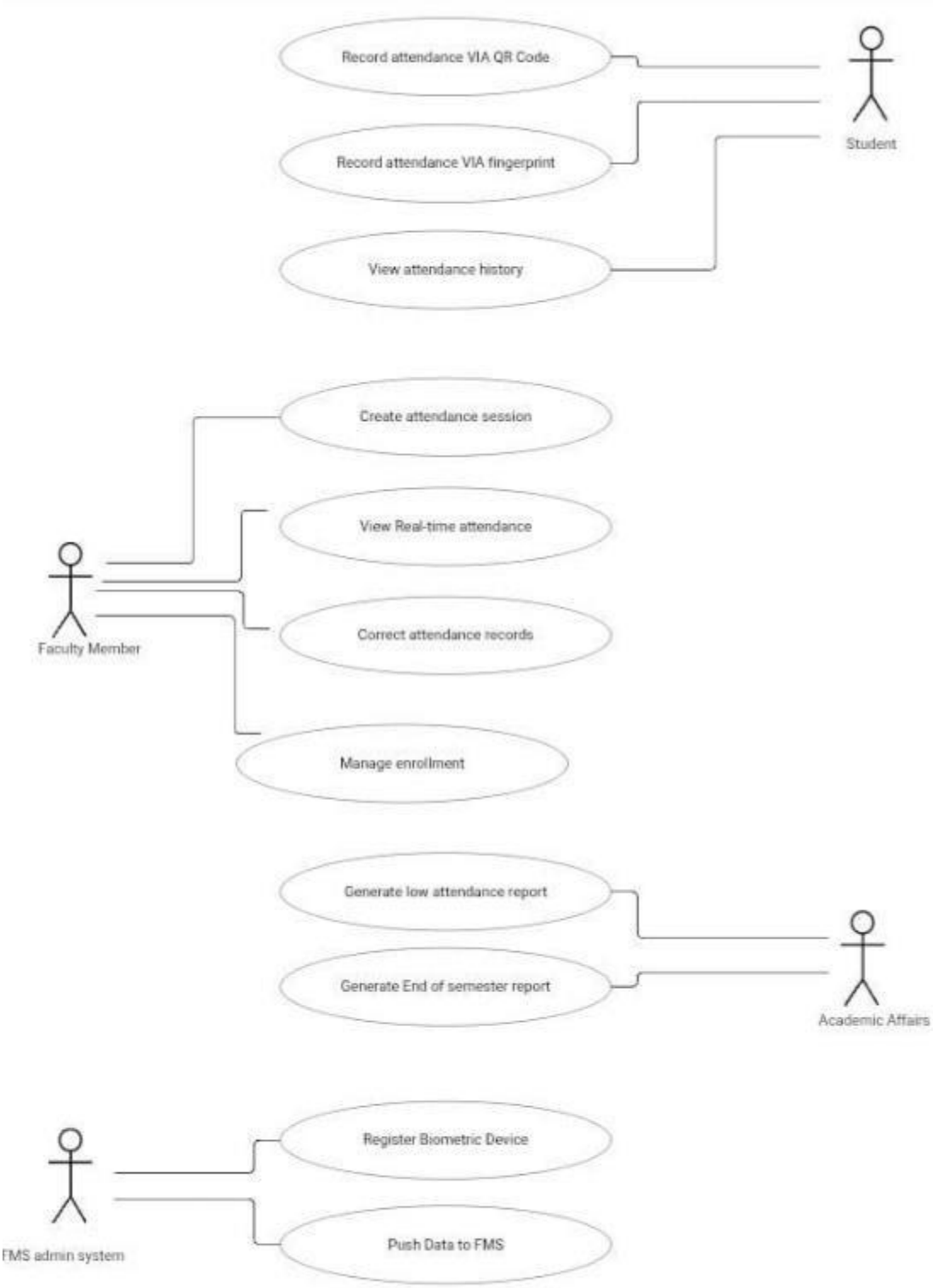
- Create Attendance Session
- View Real-time Attendance
- Correct Attendance Record
- Manage Enrollment *(New)*

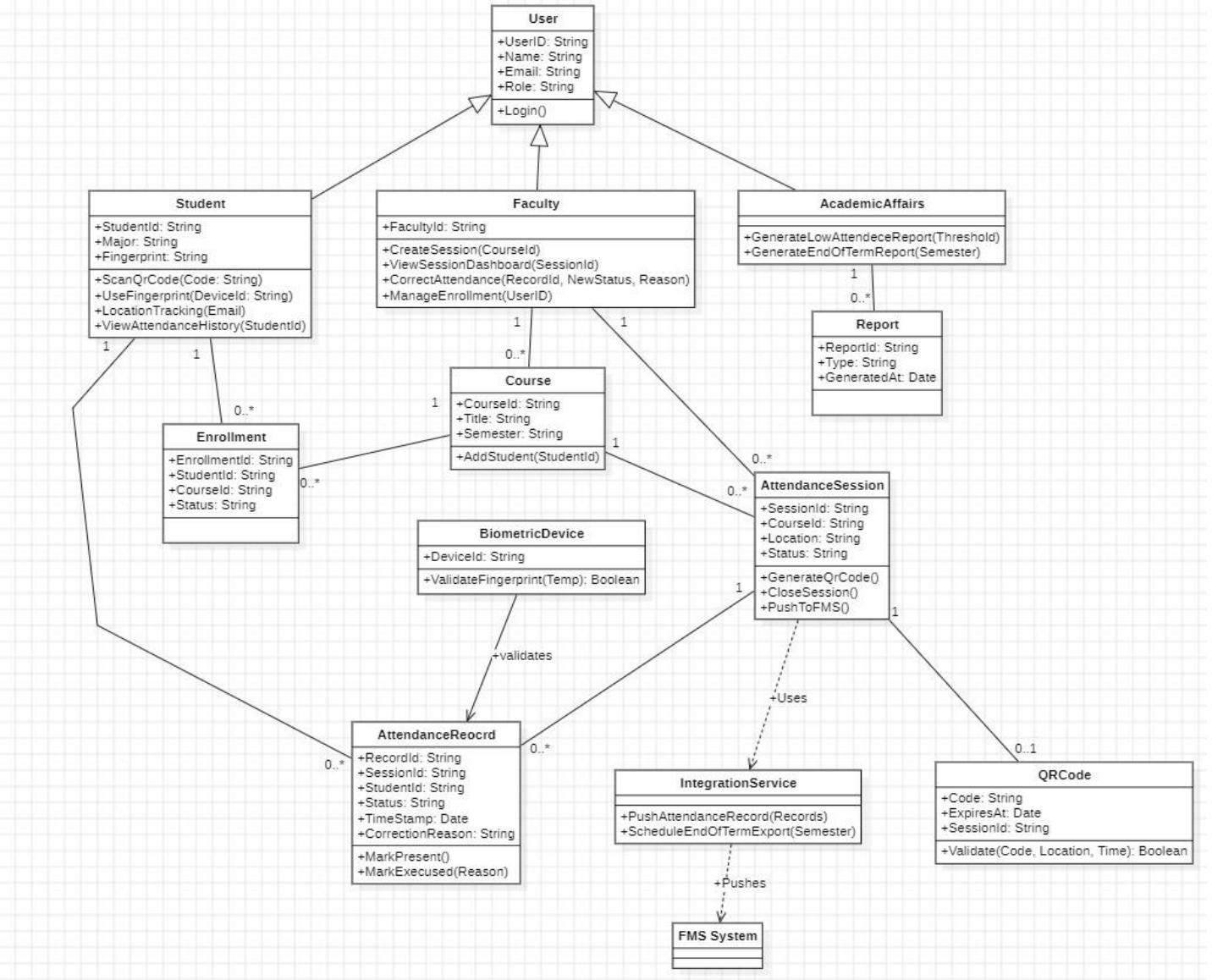
Academic Affairs

- Generate Low Attendance Report
- Generate End-of-Term Report

System / Admin

- Register Biometric Device *(New)*
- Push Data to FMS


Record attendance VIA QR Code
Record attendance VIA fingerprint
View attendance history
Student
Create attendance session
View Real-time attendance
Correct attendance records
Manage enrollment
Faculty Member
Generate low attendance report
Generate End of semester report
Academic Affairs
Register Biometric Device
Push Data to FMS
FMS admin system



Part III — State Machine Diagrams

EduTrack Attendance Management System

This part models the **dynamic lifecycle behaviour** of complex domain objects using UML State Machine Diagrams. Two objects were selected based on their rich internal behavior and the need to track state-dependent logic within the EduTrack system:

1. **LectureSession** 2. **AttendanceRecord**

Both diagrams include *composite states*, *concurrent regions*, and *internal transitions*, as encouraged in the assignment instructions.

III.c State Machine Diagram 1 — LectureSession

The **LectureSession** object contains two *parallel (concurrent) state regions* that track:

1. The **timeline** of the lecture
2. The **attendance workflow** during that lecture

Both regions operate **independently and concurrently**, and the session finishes only when **both** regions reach their final states.

Composite State: LectureInProgress

Region 1: Timeline Flow

This region models the chronological progression of a lecture:

1. **Scheduled** (initial state) ○ Represents a lecture session that is created but not yet started.
2. **InProgress** ○ Transition triggered by: **startSession** ○ Indicates the lecture is currently ongoing.
3. **Completed** ○ Transition triggered by: **endSession** ○ Marks the lecture as formally finished.

This region ends when the timeline reaches the **Completed** state.

Region 2: Attendance Flow

This region models the student attendance-taking process that can occur parallel to the lecture timeline.

1. **NotStarted** (initial state) ○ Attendance-taking has not been opened yet.
2. **Collecting** ○ Triggered by **openAttendance**
    - ○ QR scanning and fingerprint recording are accepted.
3. **Done** ○ Triggered by **closeAttendance** ○ No more attendance updates allowed.

This region ends when attendance moves into **Done**.

Completion Condition for LectureSession

The overall **LectureSession** object transitions to:

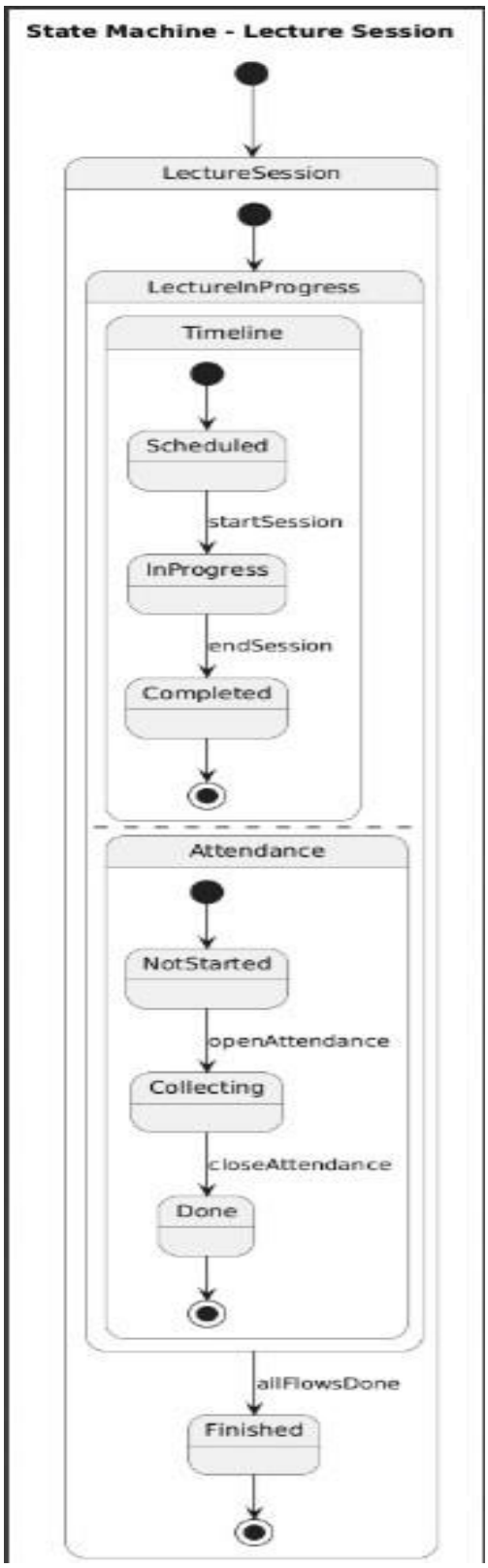
Final State: Finished

When both concurrent regions reach their final states:

- Timeline → Completed
- Attendance → Done

Transition triggered by: **allFlowsDone**

This ensures that a lecture cannot be considered fully finished until **both the lecture itself and its attendance collection are completed.**


State Machine - Lecture Session
LectureSession
LectureInProgress
Timeline
Scheduled
startSession
InProgress
endSession
Completed
Attendance
NotStarted
openAttendance
Collecting
closeAttendance
Done
allFlowsDone
Finished

III.c State Machine Diagram 2 — AttendanceRecord

The **AttendanceRecord** tracks the status of a student's attendance for a specific session. The state machine supports:

- Initial creation
- Automatic recording
- Manual correction
- Final locking of records

---

1. Initial State: Unrecorded

The student has not yet scanned the QR code nor used fingerprint verification.

Transition into next state occurs through:

- **scanAttendance()**
- **createRecord()**

→ Moves into **Recorded (composite state)**

---

2. Composite State: Recorded Once a record exists, it may be:

Present

- When student successfully scans QR or fingerprint validates
- Can transition to **Absent** through faculty correction

Absent

- When student did not scan or is marked absent
- Can transition to **Present** through correction

PresentCorrected

- A present record that has been modified by faculty
- More corrections allowed through **adjustStatus()**

AbsentCorrected

- An absent record that has been modified

- Also changeable via **adjustStatus()**

Allowed Transitions Inside "Recorded"

- correctRecord / markAbsent() Present → Absent
- correctRecord / markPresent() Absent → Present
- correctRecord / adjustStatus()
  Moves record to a corrected state variant

These transitions allow the system to keep history and enforce rules like "faculty must supply a reason for each correction."

---

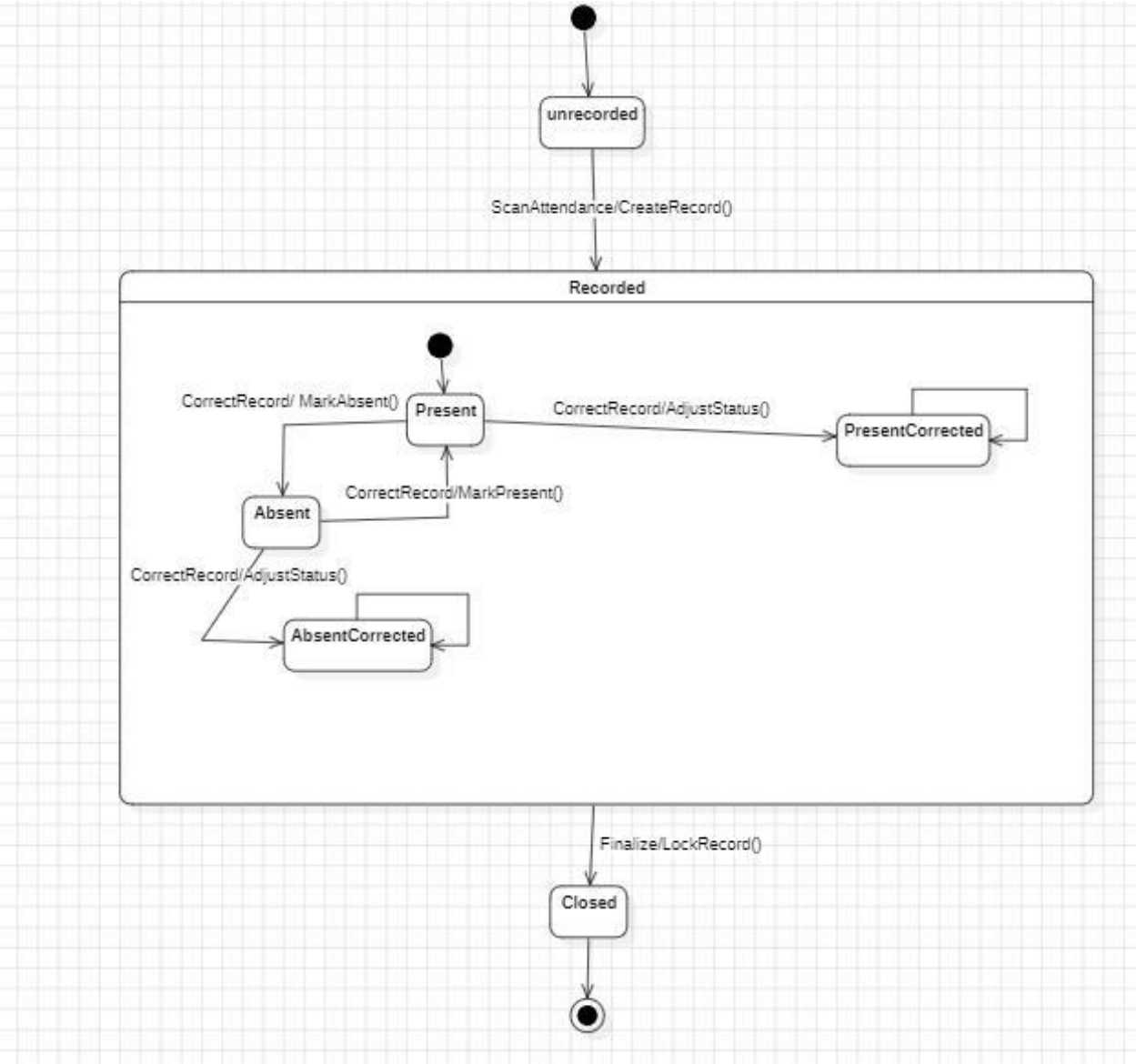
3. Final State: Closed

Triggered by:

- **finalize / lockRecord()**

At this point:

- No further edits allowed
- Used for end-of-term reporting and FMS export
- Academic Affairs gains read-only access



Use Case 1 — Record Attendance via Fingerprint

1. Use Case Description (summary)

This use case allows a student to record attendance by placing their finger on a biometric scanner. The system validates the fingerprint and either accepts or rejects the attempt.

2. Primary Scenario (Everything Goes Right)

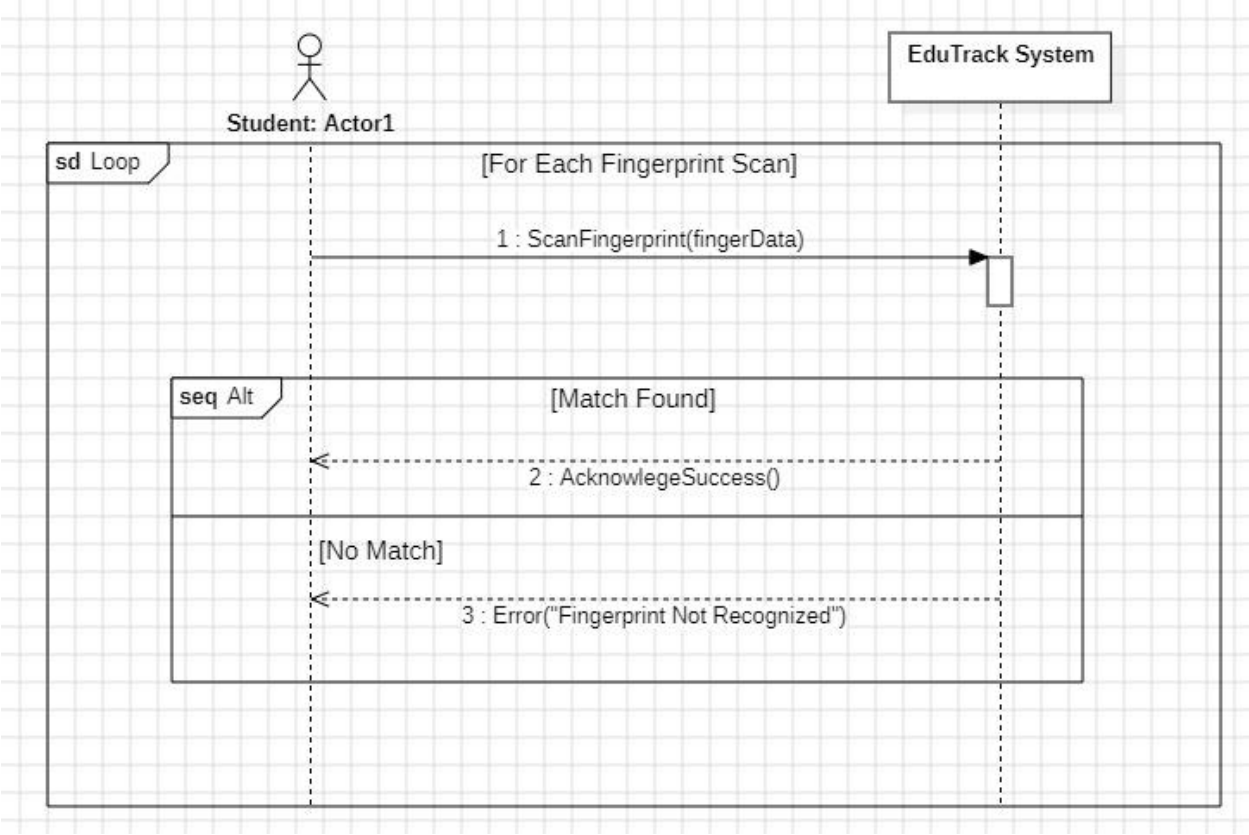
1. Student places finger on biometric scanner.
2. System receives fingerprint data.
3. System validates fingerprint and finds a matching student.
4. System records attendance.
5. System acknowledges the successful attendance recording.

3. Alternative / Exception Scenarios

A1 – Fingerprint Not Recognized

1. System cannot find a match for the scanned fingerprint.
2. System returns an error message: *"Fingerprint not recognized."* 3. Student attempts scanning again.

4. System Sequence Diagram (SSD)



5. Explanation of SSD

The SSD uses a **loop frame** to model repeated attempts and an **alt frame** to distinguish between the "match found" and "no match" scenarios. Messages clearly show the external actor (Student)

requesting system services, and the system responding with either a success acknowledgment or an error.

Use Case 2 — Create Attendance Session

1. Use Case Description (summary)

This use case allows an instructor to create a new attendance session for a course.

2. Primary Scenario

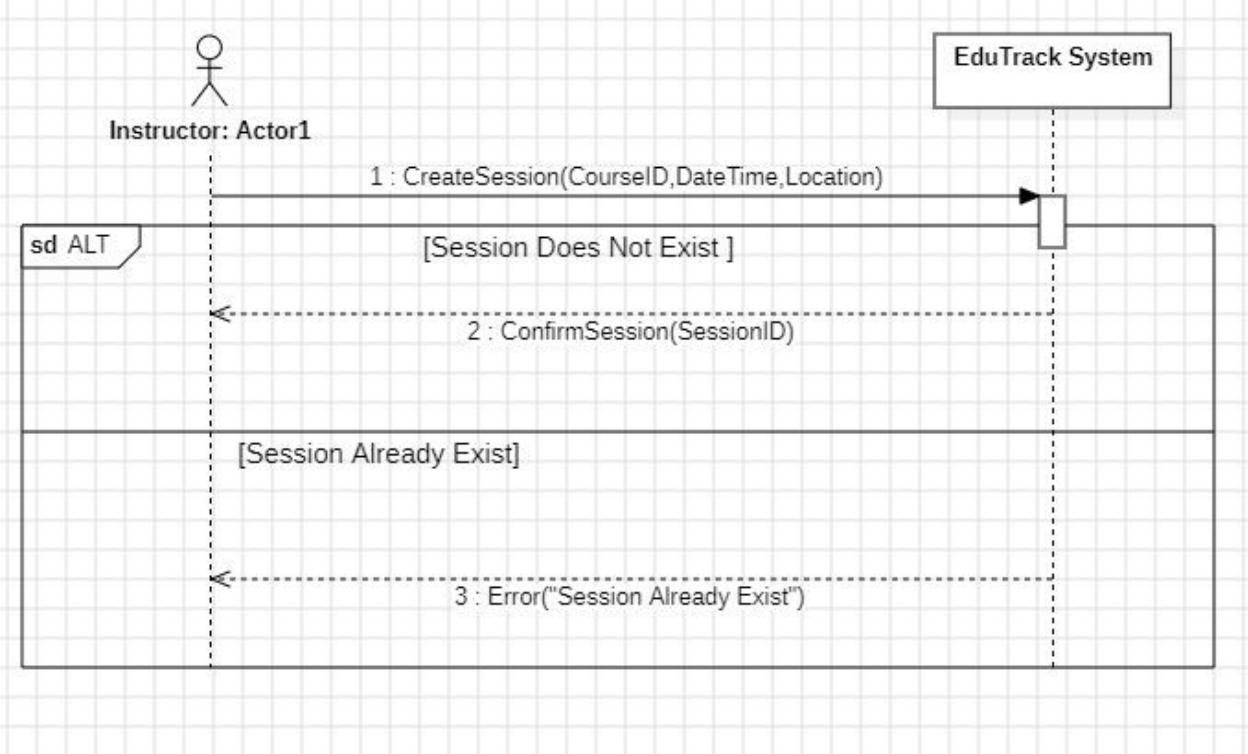
1. Instructor requests creation of a session by providing course ID, date/time, and location.
2. System checks whether a session already exists.
3. No existing session is found.
4. System creates the session.
5. System sends confirmation containing sessionID.

3. Alternative / Exception Scenario

A1 – Session Already Exists

1. System detects that a session with the same courseID and dateTime already exists.
2. System responds with an error: *"Session already exists."*

4. System Sequence Diagram (SSD)



5. Explanation of SSD

The SSD shows the instructor sending a single high-level system operation: createSession(courseID, dateTime, location). An **alt frame** cleanly separates the successful path from the exceptional one. Only system-visible operations appear, matching UML SSD principles.

Use Case 3 — Push Data to FMS

1. Use Case Description (Summary)

This use case describes how the EduTrack System sends finalized attendance data to the external **FMS System**. The operation is triggered automatically or by an administrator, and the system must handle connectivity issues, rejection, and successful synchronization.

---

2. Primary Scenario (Everything Goes Right)

1. EduTrack identifies attendance records ready for export (locked/finalized).
2. EduTrack sends a pushData(dataPackage) request to the FMS System.
3. FMS System successfully receives the data.

4. FMS validates and accepts the data.
5. FMS responds with acknowledgeSuccess().
6. EduTrack marks the records as **synced**.

---

3. Alternative / Exception Scenarios

A1 – FMS System Unavailable (No Connection)

1. EduTrack attempts to send the data.
2. No response is received or FMS is offline.
3. EduTrack logs the failure and returns an error:
   "FMS is currently unavailable."
4. System schedules a retry or requires manual re-attempt.

A2 – Data Rejected by FMS (Validation Error)

1. EduTrack pushes data to FMS.
2. FMS processes it but detects an issue (e.g., missing fields, invalid session).
3. FMS responds with error("Invalid data").
4. EduTrack logs the rejection and marks the dataset as **sync failed**.
5. Admin may correct records or regenerate the data package.

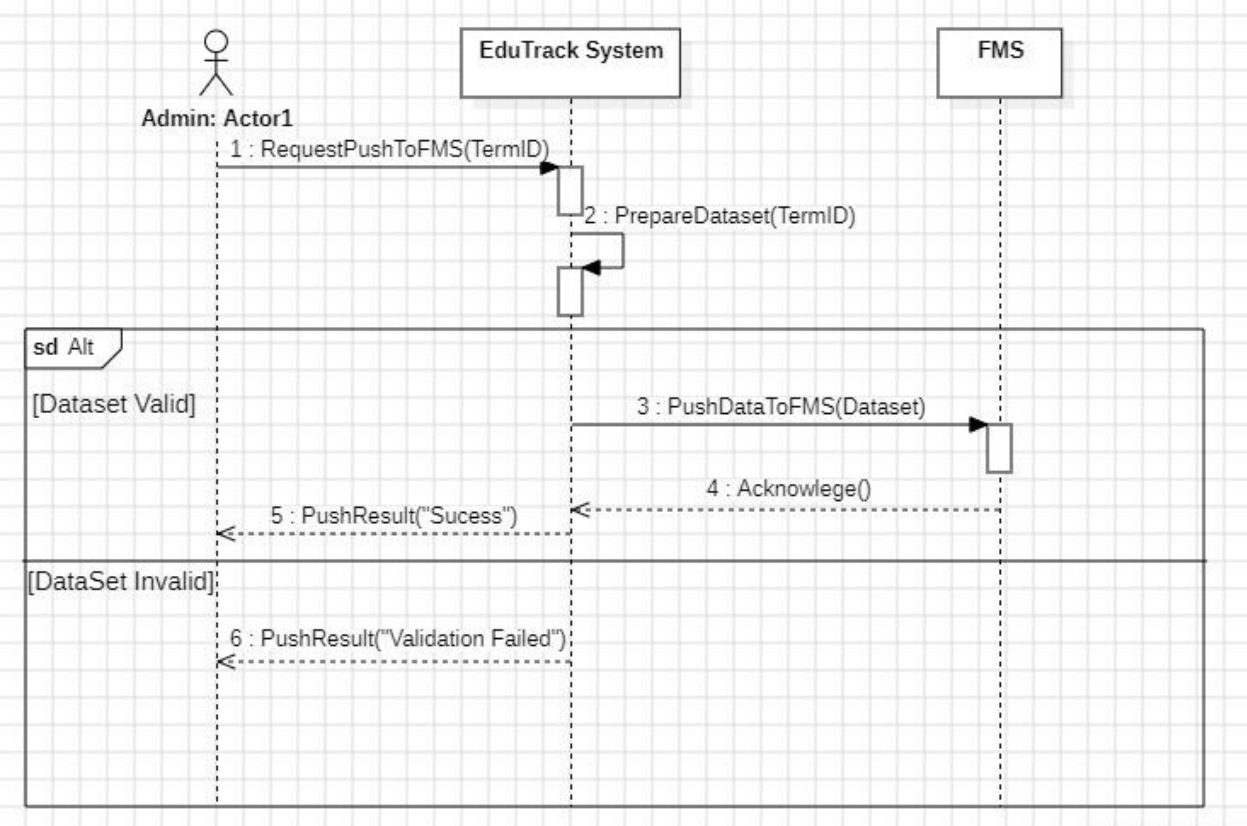
A3 – Partial Success

If the external FMS accepts some records but rejects others:

1. FMS returns a detailed error list.
2. EduTrack marks accepted records as synced.
3. Remaining records stay unsynced for correction.

---

4. System Sequence Diagram (SSD)



---

5. Explanation of SSD

This SSD models how EduTrack interacts with the external FMS System to transfer attendance data. A single system operation (pushData) is shown, maintaining SSD principles. The **alt frame** divides behavior into three branches: successful synchronization, FMS unavailability, and FMS rejecting invalid data. This separation keeps the diagram clear while representing all major exception paths. The messages shown are system-visible events only, which complies with UML SSD guidelines.