Campus Access Platform

An IIT campus has multiple secure entry points:

- Hostel gates
- Lab doors
- Library turnstiles
- Sports complex
- Exam hall entry

People on campus use different identity methods:

- RFID card
- QR code on mobile app
- Biometric fingerprint
- Face ID (optional)

Different zones have different rules:

- Library: must have active membership + no pending fines
- Labs: only specific departments + time windows
- Exam halls: only students with valid admit card + correct seat
- Hostels: hostel residents only after 10 PM

The system must validate identity and rules and log access attempts.

Problem Statement

A university campus has multiple secure access zones such as **Library, Labs, Hostel Gates, and Exam Halls**. People enter these zones using different authentication methods such as **RFID cards, QR codes, and Biometrics**. Each zone has different access rules, and the system must validate and decide whether entry is allowed.

Your task is to build a **modular Access Control System** using **interfaces**, so that:

- New authentication methods can be added without changing the core workflow.
- New zone policies can be added without rewriting existing code.
- Logging and notifications can be plugged in later.

Entities (must create as classes)

1. **Person**
    - `id`, `name`
    - `role` (STUDENT / STAFF / VISITOR)
    - `department` (e.g., CSE, EE, ME)

- `hostelId` (nullable)
- `finesDue` (double)

2. **Zone**
    - `zoneId`, `name`
    - `zoneType` (LIBRARY / LAB / HOSTEL / EXAM_HALL)
    - `openHour`, `closeHour` (0–23)
3. **AccessRequest**
    - `Person person`
    - `Zone zone`
    - `LocalDateTime timestamp`
    - `AuthType authType` (RFID / QR / BIOMETRIC)
    - `String credentialData` (the scanned input)
4. **AccessResult**
    - `boolean allowed`
    - `String reason`
    - `LocalDateTime timestamp`

Interfaces (must implement)

1) Authenticator

```
boolean authenticate(AccessRequest request);

AuthType supportedType();
```

Implementations required:

- `RfidAuthenticator`
- `QrAuthenticator`
- `BiometricAuthenticator`

2) AccessPolicy

```
boolean supports(Zone zone);

AccessResult evaluate(AccessRequest request);
```

Implementations required:

- `LibraryPolicy`
- `LabPolicy`
- `HostelPolicy`
- `ExamHallPolicy`

3) AuditLogger

```
void log(AccessRequest request, AccessResult result);
```

Implementation required:

- `ConsoleAuditLogger`

4) Notifier

```
void notify(AccessRequest request, AccessResult result);
```

Implementation required:

- `ConsoleNotifier`

Zone Rules (must enforce)

LibraryPolicy

- Deny if role == VISITOR
- Deny if finesDue > 0
- Deny if outside zone open/close hours

LabPolicy

- Allow only if person.department matches zone department label stored in zone name (example: “CSE Lab” means only CSE)
- Deny if outside open/close hours

HostelPolicy

- If time >= 22:00, allow only if person.hostelId matches zone hostelId encoded in zone name (example: “Hostel-H1 Gate” means H1)
- Otherwise allow (before 10 PM)

ExamHallPolicy

- Allow only STUDENT
- credentialData must contain a valid exam token format: `"EXAM:<personId>"`
    - Example valid token for personId=101: `EXAM:101`

Main Workflow Requirement (most important)

Create a central class `AccessControlService` that:

1. Chooses correct `Authenticator` based on request.authType
2. Authenticates
3. Chooses correct `AccessPolicy` based on zone
4. Evaluates access
5. Logs result using `AuditLogger`
6. Sends notification using `Notifier`
7. Prints final decision

✅ **Hard requirement:** You must NOT use a giant `if-else` chain for every zone + auth combination inside the service. You must use interfaces and polymorphism.

Expected Output (example)

```
[AUTH] QR authenticated for Person 101

[POLICY] LIBRARY policy evaluated: DENIED (Fines due: 250.0)

[AUDIT] 2026-02-11T10:15:21 Person=101 Zone=Central Library
Allowed=false Reason=Fines due

[NOTIFY] Access denied for Central Library. Reason: Fines due

FINAL: DENIED
```

Detailed Step-by-Step Solution

Step 1: Create enums (Role, ZoneType, AuthType)

Why

Enums:

- prevent invalid string values like `"studnt"` or `"LIBRRY"`
- make switch/selection safer
- improve readability in output and debugging

Code

Role.java

```
enum Role {
    STUDENT, STAFF, VISITOR
}
```

ZoneType.java

```
enum ZoneType {
    LIBRARY, LAB, HOSTEL, EXAM_HALL
}
```

AuthType.java

```
enum AuthType {
    RFID, QR, BIOMETRIC
}
```

Step 2: Create core domain classes (Person, Zone, AccessRequest, AccessResult)

Why

These are the “data carriers” of your system:

- Policies and authenticators should read inputs from these objects
- Service should not pass raw strings around

Design rules (important)

- Fields **private + final** (immutability prevents bugs)
- Provide **constructor + getters**
- Keep domain classes small and focused
- `AccessResult` uses factory methods: `allow()` / `deny()` for clean readable code

---

2A) `Person.java`

```
import java.util.Objects;

final class Person {
    private final int id;
    private final String name;
    private final Role role;
    private final String department;
    private final String hostelId; // can be null
    private final double finesDue;

    public Person(int id, String name, Role role, String department,
String hostelId, double finesDue) {
        this.id = id;
```

```
        this.name = Objects.requireNonNull(name);
        this.role = Objects.requireNonNull(role);
        this.department = Objects.requireNonNull(department);
        this.hostelId = hostelId;
        this.finesDue = finesDue;
    }

    public int getId() { return id; }
    public String getName() { return name; }
    public Role getRole() { return role; }
    public String getDepartment() { return department; }
    public String getHostelId() { return hostelId; }
    public double getFinesDue() { return finesDue; }
}
```

2B) Zone.java

```
import java.util.Objects;

final class Zone {
    private final String zoneId;
    private final String name;
    private final ZoneType zoneType;
    private final int openHour;
    private final int closeHour;

    public Zone(String zoneId, String name, ZoneType zoneType, int
openHour, int closeHour) {
        this.zoneId = Objects.requireNonNull(zoneId);
        this.name = Objects.requireNonNull(name);
        this.zoneType = Objects.requireNonNull(zoneType);
        this.openHour = openHour;
        this.closeHour = closeHour;
    }

    public String getZoneId() { return zoneId; }
    public String getName() { return name; }
    public ZoneType getZoneType() { return zoneType; }
    public int getOpenHour() { return openHour; }
    public int getCloseHour() { return closeHour; }
```

```
}
```

2C) AccessRequest.java

```
import java.time.LocalDateTime;
import java.util.Objects;

final class AccessRequest {
    private final Person person;
    private final Zone zone;
    private final LocalDateTime timestamp;
    private final AuthType authType;
    private final String credentialData;

    public AccessRequest(Person person, Zone zone, LocalDateTime
timestamp, AuthType authType, String credentialData) {
        this.person = Objects.requireNonNull(person);
        this.zone = Objects.requireNonNull(zone);
        this.timestamp = Objects.requireNonNull(timestamp);
        this.authType = Objects.requireNonNull(authType);
        this.credentialData =
Objects.requireNonNull(credentialData);
    }

    public Person getPerson() { return person; }
    public Zone getZone() { return zone; }
    public LocalDateTime getTimestamp() { return timestamp; }
    public AuthType getAuthType() { return authType; }
    public String getCredentialData() { return credentialData; }
}
```

2D) AccessResult.java

```
import java.time.LocalDateTime;
import java.util.Objects;

final class AccessResult {
    private final boolean allowed;
```

```
    private final String reason;
    private final LocalDateTime timestamp;

    private AccessResult(boolean allowed, String reason,
LocalDateTime timestamp) {
        this.allowed = allowed;
        this.reason = Objects.requireNonNull(reason);
        this.timestamp = Objects.requireNonNull(timestamp);
    }

    public static AccessResult allow(String reason, LocalDateTime
timestamp) {
        return new AccessResult(true, reason, timestamp);
    }

    public static AccessResult deny(String reason, LocalDateTime
timestamp) {
        return new AccessResult(false, reason, timestamp);
    }

    public boolean isAllowed() { return allowed; }
    public String getReason() { return reason; }
    public LocalDateTime getTimestamp() { return timestamp; }
}
```

Step 3: Define the `Authenticator` interface

Why interface

We have multiple authentication methods. We want the service to call:

- `authenticate()` no matter what method it is
- without changing service code for each new method

This is **polymorphism**: same call, different behavior.

Code: `Authenticator.java`

```
interface Authenticator {
    AuthType supportedType();
    boolean authenticate(AccessRequest request);
}
```

Step 4: Implement Authenticators (RFID / QR / Biometric)

Why

Each authenticator:

- validates credentialData format differently
- simulates real hardware/SDK integration

Simulated rules

- RFID: `"RFID:<personId>"`
- QR: `"QR:<personId>"`
- BIO: `"BIO:<personId>:OK"`

`RfidAuthenticator.java`

```
final class RfidAuthenticator implements Authenticator {
    @Override
    public AuthType supportedType() { return AuthType.RFID; }

    @Override
    public boolean authenticate(AccessRequest request) {
        String expected = "RFID:" + request.getPerson().getId();
```

```
        return
request.getCredentialData().trim().equalsIgnoreCase(expected);
    }
}
```

QrAuthenticator.java

```
final class QrAuthenticator implements Authenticator {
    @Override
    public AuthType supportedType() { return AuthType.QR; }

    @Override
    public boolean authenticate(AccessRequest request) {
        String expected = "QR:" + request.getPerson().getId();
        return
request.getCredentialData().trim().equalsIgnoreCase(expected);
    }
}
```

BiometricAuthenticator.java

```
final class BiometricAuthenticator implements Authenticator {
    @Override
    public AuthType supportedType() { return AuthType.BIOMETRIC; }

    @Override
    public boolean authenticate(AccessRequest request) {
        String expected = "BIO:" + request.getPerson().getId() +
":OK";
        return
request.getCredentialData().trim().equalsIgnoreCase(expected);
    }
}
```

Step 5: Define the `AccessPolicy` interface

Why interface

Zones have different rules. We need:

- a common contract: `evaluate(request)`
- a selection mechanism: `supports(zone)`

So the service can pick the correct policy dynamically.

Code: `AccessPolicy.java`

```
interface AccessPolicy {
    boolean supports(Zone zone);
    AccessResult evaluate(AccessRequest request);
}
```

Step 6: Create reusable policy helper utilities

Why

Multiple policies need:

- time checking
- parsing zone name tokens (dept / hostel id)

To avoid duplication, we centralize it.

Code: `PolicyUtils.java`

```
import java.time.LocalDateTime;

final class PolicyUtils {
    private PolicyUtils() {}

    static boolean isWithinHours(Zone zone, LocalDateTime t) {
```

```
        int h = t.getHour();
        return h >= zone.getOpenHour() && h < zone.getCloseHour();
    }

    static String leadingToken(String zoneName) {
        String[] parts = zoneName.trim().split("\\s+");
        return parts.length > 0 ? parts[0].trim() : "";
    }

    static String hostelFromName(String zoneName) {
        int idx = zoneName.indexOf("Hostel-");
        if (idx < 0) return null;

        int start = idx + "Hostel-".length();
        int end = zoneName.indexOf(' ', start);
        if (end < 0) end = zoneName.length();

        return zoneName.substring(start, end).trim();
    }
}
```

Step 7: Implement Policies (Library, Lab, Hostel, Exam Hall)

Why

Each policy:

- encapsulates its own rules
- is replaceable or extendable
- makes service code stable

7A) `LibraryPolicy.java`

Rules:

- deny if closed
- deny visitors
- deny if fines > 0

```
import java.time.LocalDateTime;

final class LibraryPolicy implements AccessPolicy {
    @Override
    public boolean supports(Zone zone) {
        return zone.getZoneType() == ZoneType.LIBRARY;
    }

    @Override
    public AccessResult evaluate(AccessRequest request) {
        Zone z = request.getZone();
        Person p = request.getPerson();
        LocalDateTime t = request.getTimestamp();

        if (!PolicyUtils.isWithinHours(z, t)) {
            return AccessResult.deny("Library closed (outside
allowed hours)", t);
        }
        if (p.getRole() == Role.VISITOR) {
            return AccessResult.deny("Visitors not allowed in
library", t);
        }
        if (p.getFinesDue() > 0) {
            return AccessResult.deny("Fines due: " +
p.getFinesDue(), t);
        }
        return AccessResult.allow("Library access granted", t);
    }
}
```

7B) `LabPolicy.java`

Rules:

- deny if closed
- allow only matching department (zone name starts with dept)

```
import java.time.LocalDateTime;

final class LabPolicy implements AccessPolicy {
    @Override
    public boolean supports(Zone zone) {
        return zone.getZoneType() == ZoneType.LAB;
    }

    @Override
    public AccessResult evaluate(AccessRequest request) {
        Zone z = request.getZone();
        Person p = request.getPerson();
        LocalDateTime t = request.getTimestamp();

        if (!PolicyUtils.isWithinHours(z, t)) {
            return AccessResult.deny("Lab closed (outside allowed
hours)", t);
        }

        String requiredDept = PolicyUtils.leadingToken(z.getName());
// "CSE" from "CSE Lab"
        if (!requiredDept.equalsIgnoreCase(p.getDepartment())) {
            return AccessResult.deny(
                    "Department mismatch. Required: " + requiredDept
+ ", Yours: " + p.getDepartment(), t);
        }

        return AccessResult.allow("Lab access granted", t);
    }
}
```

7C) `HostelPolicy.java`

Rules:

- after 10 PM (>=22): allow only if hostel matches

```
import java.time.LocalDateTime;

final class HostelPolicy implements AccessPolicy {
    @Override
    public boolean supports(Zone zone) {
        return zone.getZoneType() == ZoneType.HOSTEL;
    }

    @Override
    public AccessResult evaluate(AccessRequest request) {
        Zone z = request.getZone();
        Person p = request.getPerson();
        LocalDateTime t = request.getTimestamp();

        if (t.getHour() >= 22) {
            String requiredHostel =
PolicyUtils.hostelFromName(z.getName()); // "H1" from "Hostel-H1
Gate"
            if (requiredHostel == null) {
                return AccessResult.deny("Hostel zone
misconfigured", t);
            }
            if (p.getHostelId() == null ||
!requiredHostel.equalsIgnoreCase(p.getHostelId())) {
                return AccessResult.deny("Hostel restricted after 10
PM. Required hostel: " + requiredHostel, t);
            }
        }

        return AccessResult.allow("Hostel access granted", t);
    }
}
```

7D) `ExamHallPolicy.java`

Rules:

- only STUDENT
- token must be `EXAM:<personId>` (uses credentialData)

```
import java.time.LocalDateTime;

final class ExamHallPolicy implements AccessPolicy {
    @Override
    public boolean supports(Zone zone) {
        return zone.getZoneType() == ZoneType.EXAM_HALL;
    }

    @Override
    public AccessResult evaluate(AccessRequest request) {
        Person p = request.getPerson();
        LocalDateTime t = request.getTimestamp();

        if (p.getRole() != Role.STUDENT) {
            return AccessResult.deny("Only students allowed in exam
hall", t);
        }

        String expected = "EXAM:" + p.getId();
        if
(!request.getCredentialData().trim().equalsIgnoreCase(expected)) {
            return AccessResult.deny("Invalid exam token", t);
        }

        return AccessResult.allow("Exam hall access granted", t);
    }
}
```

Step 8: Create interfaces for logging and notification

Why

Real systems evolve:

- logging may go to DB, file, SIEM
- notifications may go to SMS/email/app

Interfaces allow easy replacement.

AuditLogger.java

```
interface AuditLogger {
    void log(AccessRequest request, AccessResult result);
}
```

Notifier.java

```
interface Notifier {
    void notify(AccessRequest request, AccessResult result);
}
```

Step 9: Implement Console logger and notifier

ConsoleAuditLogger.java

```
final class ConsoleAuditLogger implements AuditLogger {
    @Override
    public void log(AccessRequest request, AccessResult result) {
        System.out.println("[AUDIT] " + result.getTimestamp()
                + " Person=" + request.getPerson().getId()
                + " Zone=" + request.getZone().getName()
                + " Allowed=" + result.isAllowed()
                + " Reason=" + result.getReason());
    }
}
```

ConsoleNotifier.java

```
final class ConsoleNotifier implements Notifier {
    @Override
    public void notify(AccessRequest request, AccessResult result) {
        if (result.isAllowed()) {
            System.out.println("[NOTIFY] Access granted for " +
request.getZone().getName());
        } else {
            System.out.println("[NOTIFY] Access denied for " +
request.getZone().getName() +
                    ". Reason: " + result.getReason());
        }
    }
}
```

Step 10: Implement the core workflow service (MOST IMPORTANT)

Why this step is the whole point

`AccessControlService` must:

- depend only on interfaces
- choose correct authenticator/policy at runtime
- remain unchanged when adding new implementations

This is where students see **the actual value of interfaces**.

AccessControlService.java

```
import java.time.LocalDateTime;
import java.util.List;
import java.util.Objects;
```

```
final class AccessControlService {
    private final List<Authenticator> authenticators;
    private final List<AccessPolicy> policies;
    private final AuditLogger auditLogger;
    private final Notifier notifier;

    public AccessControlService(List<Authenticator> authenticators,
                                List<AccessPolicy> policies,
                                AuditLogger auditLogger,
                                Notifier notifier) {
        this.authenticators =
List.copyOf(Objects.requireNonNull(authenticators));
        this.policies =
List.copyOf(Objects.requireNonNull(policies));
        this.auditLogger = Objects.requireNonNull(auditLogger);
        this.notifier = Objects.requireNonNull(notifier);
    }

    public AccessResult process(AccessRequest request) {
        // 1) Choose authenticator
        Authenticator auth =
findAuthenticator(request.getAuthType());
        if (auth == null) {
            AccessResult r = AccessResult.deny("No authenticator for
" + request.getAuthType(), LocalDateTime.now());
            finalizeIO(request, r);
            return r;
        }

        // 2) Authenticate
        boolean ok = auth.authenticate(request);
        System.out.println("[AUTH] " + request.getAuthType() + " =>
" + (ok ? "OK" : "FAIL"));
        if (!ok) {
            AccessResult r = AccessResult.deny("Authentication
failed", request.getTimestamp());
            finalizeIO(request, r);
            return r;
        }

        // 3) Choose policy
```

```
        AccessPolicy policy = findPolicy(request.getZone());
        if (policy == null) {
            AccessResult r = AccessResult.deny("No policy for zone
type " + request.getZone().getZoneType(), request.getTimestamp());
            finalizeIO(request, r);
            return r;
        }

        // 4) Evaluate
        AccessResult result = policy.evaluate(request);
        System.out.println("[POLICY] " +
request.getZone().getZoneType()
                + " => " + (result.isAllowed() ? "ALLOWED" :
"DENIED")
                + " (" + result.getReason() + ")");

        // 5) Log + notify
        finalizeIO(request, result);
        return result;
    }

    private void finalizeIO(AccessRequest req, AccessResult res) {
        auditLogger.log(req, res);
        notifier.notify(req, res);
        System.out.println("FINAL: " + (res.isAllowed() ? "ALLOWED"
: "DENIED"));

System.out.println("-----------------------------------------------
----");
    }

    private Authenticator findAuthenticator(AuthType type) {
        for (Authenticator a : authenticators) {
            if (a.supportedType() == type) return a;
        }
        return null;
    }

    private AccessPolicy findPolicy(Zone zone) {
        for (AccessPolicy p : policies) {
            if (p.supports(zone)) return p;
```

```
        }
        return null;
    }
}
```

Step 11: Main class to demonstrate the system

Why

Students must see the system working end-to-end, and observe different results.

Main.java

```
import java.time.LocalDateTime;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        Person student = new Person(101, "Suhani", Role.STUDENT, "CSE",
"H1", 0);
        Person fineStudent = new Person(102, "FineStudent", Role.STUDENT,
"CSE", "H2", 250);
        Person staff = new Person(201, "Vishal", Role.STAFF, "ME", null,
0);
        Person visitor = new Person(999, "Visitor", Role.VISITOR, "NA",
null, 0);

        Zone library = new Zone("Z1", "Central Library", ZoneType.LIBRARY,
9, 20);
        Zone cseLab = new Zone("Z2", "CSE Lab", ZoneType.LAB, 10, 18);
        Zone hostelH1 = new Zone("Z3", "Hostel-H1 Gate", ZoneType.HOSTEL,
0, 24);
        Zone examHall = new Zone("Z4", "Exam Hall A", ZoneType.EXAM_HALL,
8, 17);

        AccessControlService service = new AccessControlService(
```

```
                List.of(new RfidAuthenticator(), new QrAuthenticator(), new
BiometricAuthenticator()),
                List.of(new LibraryPolicy(), new LabPolicy(), new
HostelPolicy(), new ExamHallPolicy()),
                new ConsoleAuditLogger(),
                new ConsoleNotifier()
        );

        LocalDateTime now = LocalDateTime.now();

        service.process(new AccessRequest(student, library,
now.withHour(11), AuthType.QR, "QR:101"));         // allow
        service.process(new AccessRequest(fineStudent, library,
now.withHour(11), AuthType.RFID, "RFID:102")); // deny fines
        service.process(new AccessRequest(visitor, library,
now.withHour(11), AuthType.QR, "QR:999"));         // deny visitor
        service.process(new AccessRequest(staff, cseLab, now.withHour(12),
AuthType.BIOMETRIC, "BIO:201:OK")); // deny dept
        service.process(new AccessRequest(student, cseLab,
now.withHour(12), AuthType.BIOMETRIC, "BIO:101:OK"));// allow
        service.process(new AccessRequest(fineStudent, hostelH1,
now.withHour(22), AuthType.QR, "QR:102"));    // deny hostel mismatch
        service.process(new AccessRequest(student, examHall,
now.withHour(9), AuthType.QR, "EXAM:101"));       // allow
        service.process(new AccessRequest(student, examHall,
now.withHour(9), AuthType.QR, "EXAM:999"));       // deny token
    }
}
```

“Why interface?” — interview-ready answer

> I use interfaces here because authentication methods, zone policies, notification channels, and logging destinations are independent behaviors that change frequently. Interfaces allow me to program to abstractions, swap implementations, add new capabilities without modifying core workflow code, and make the system testable by mocking these dependencies.