a)

We used a basic client server architecture as our system Architectural pattern to ensure that communication is simple between the frontend and the backend while making sure the system separates between them so it doesn’t become hard to read and maintain.

We also used Notification/Publisher-Notification Architectural pattern as our notification architectural pattern because it ensures scalable, flexible, and event-driven communication tailored to diverse alumni interests while maintaining centralized tracking for compliance and reporting.

c)  
A login interface : User

A signup/register interface : user (students/alumni)   
An interface for viewing/verifying unverified user : Faculty Staff   
An event scheduling form interface : Faculty Staff

An interface showing events and linking them : Alumni and Faculty Staff  
An event showcasing interface with ability to join the event (Faculty Staff can view participants): User   
A messaging/notifications interface : Alumni   
A profile interface : User   
A mentorship application interface and show current students: Alumni   
A mentorship view with ability for students to apply : Student A profile page with ability to edit user data : User

A donation form/history interface : Alumni   
A newsletter drafting/publishing interface : Faculty Staff

A page showcasing currently drafted newsletter with the ability to edit/publish them : Faculty Staff   
A newsletter showcase interface : User

An interface with search results : User

An index page : user

A page viewing all donation history : Faculty Staff

An event rescheduling interface : Faculty Staff   
Admin panel : Admin

H)  
We used Actor class as it’s easy to represent how it works and it’s easier to read and change it for most people even though it has limited reusability

j)

We used **State pattern** , **Immutable Pattern**, and **Observer pattern**   
**State pattern:**

State is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.  
It replaces the complex conditionals with a simple object state that dictates the object’s behavior.  
This pattern improves readability and maintainability especially when classes have simple state that it goes through

Used in newsletter class where there are 2 states published state and draft state  
  
**Immutable pattern:**   
An immutable object is an object that has a state that never changes after creation.  
It helps with creating classes that have values that never change.  
It makes the values of some objects stay the same without changing.

Used in Donation class where Alumni constructs the class when he donates but it is never changed again.  
  
**Observer pattern**:

The Observer pattern defines a one-to-many dependency so that when once the subject changes state (event), all its dependents (subscribers) are notified automatically.  
It helps with sending notifications to users who are subscribed without sending to users who have no interest in the event.  
It changes the classes and notifications structure of the whole system.

Alumni users subscribe to Newsletter and Event notifications (Subjects), which broadcast alerts to all registered observers when fresh content becomes available.

l)

God Object (aka “Blob”): a class that “knows too much or does too much,” gathering unrelated methods and data, and becoming tightly coupled to many parts of the system

The Faculty Staff class handles:

Account verification and user management

Event scheduling and participant tracking

Newsletter drafting/publishing

Viewing donation reports

Generating event reports

This concentration of duties makes it difficult to change any one feature without risking side-effects elsewhere

How to Avoid and Refactor This Smell

1.Applying the Single-Responsibility Principle (SRP)

Ensure each class has one, and only one, reason to change

2.Group Related Methods and Data

Move static utility methods into dedicated utility classes.

m)

| Alumni | Entity |
| --- | --- |
| Student | Entity |
| Admin | Entity |
| User (abstract) | Entity |
| FacultyStaff | Entity |
| Event | Entity |
| Newsletter | Entity |
| Mentorship | Entity |
| Donation | Entity |
| Subject | Control |
| Observer | Control |
| State (interface) | Application Logic |
| DraftState | Application Logic |
| PublishedState | Application Logic |

n)  
No, I did not rely on forks or cascades in my interaction diagrams. Instead, I used straightforward sequential interactions between objects to maintain clarity and simplicity. Reason: The functionality I modeled did not require concurrent behaviors (which would need forks) or complex message chaining (which would suggest cascades). The interactions were linear and easy to represent using standard message flows.

q)

### **1. User**

| **Column** | **Type** | **Constraints** |
| --- | --- | --- |
| user\_id | INT | PK |
| username | VARCHAR(100) | NOT NULL, UNIQUE |
| password\_hash | VARCHAR(500) | NOT NULL |
| role | VARCHAR(50) | ENUM('Alumni', 'Student', 'Admin', 'FacultyStaff') |

### 

### **2. Alumni**

| **Column** | **Type** | **Constraints** |
| --- | --- | --- |
| userId | INT | PK, FK → User(user\_id) |
| mentor | BOOLEAN | NOT NULL DEFAULT FALSE |
| verified | BOOLEAN | NOT NULL DEFAULT FALSE |
| graduationDate | DATE | CHECK (graduationDate <= CURDATE()) |
| major | VARCHAR(100) |  |

### 

### **3. Student**

| **Column** | **Type** | **Constraints** |
| --- | --- | --- |
| userId | INT | PK, FK → User(user\_id) |
| major | VARCHAR(100) |  |

### 

### **4. FacultyStaff**

| **Column** | **Type** | **Constraints** |
| --- | --- | --- |
| user\_id | INT | PK, FK → User(user\_id) |

### **5. Admin**

| **Column** | **Type** | **Constraints** |
| --- | --- | --- |
| user\_id | INT | PK, FK → User(user\_id) |

### 

### **6. Mentorship**

| **Column** | **Type** | **Constraints** |
| --- | --- | --- |
| mentorship\_id | INT | PK |
| mentor\_id | INT | FK → Alumni(user\_id)  NOT NULL |
| description | TEXT | NOT NULL |
| date | DATE | CHECK (date <= CURDATE()) |

### 

### **7. Event**

| **Column** | **Type** | **Constraints** |
| --- | --- | --- |
| eventId | INT | PK |
| name | VARCHAR(255) | NOT NULL |
| description | TEXT |  |
| date | DATE | NOT NULL CHECK (date <= CURDATE()) |
| creatorId | INT | FK → FacultyStaff(user\_id) |

### 

### 

### **8. EventParticipant**

| **Column** | **Type** | **Constraints** |
| --- | --- | --- |
| event\_id | INT | PK part, FK → Event(event\_id) |
| participant\_id | INT | PK part, FK → Alumni(user\_id) |
| RSVP\_status | BOOLEAN | DEFAULT FALSE |

### 

### **9. Donation**

| **Column** | **Type** | **Constraints** |
| --- | --- | --- |
| donation\_id | INT | PK |
| amount | DECIMAL(10,2) | NOT NULL, CHECK(amount>0) |
| cause | VARCHAR(255) |  |
| date | DATE | CHECK (date <= CURDATE()) |
| donor\_id | INT | FK → Alumni(user\_id) NOT NULL |

### 

### **10. Newsletter**

| **Column** | **Type** | **Constraints** |
| --- | --- | --- |
| newsletter\_id | INT | PK |
| title | VARCHAR(255) | NOT NULL |
| body | TEXT | NOT NULL |
| creatorId | INT | FK → FacultyStaff(user\_id) NOT NULL |
| publishedState | BOOLEAN | DEFAULT FALSE |

### **11. Notification**

| **Column** | **Type** | **Constraints** |
| --- | --- | --- |
| notification\_id | INT | PK |
| notification | TEXT | NOT NULL |
| user\_id | INT | PK, FK → User(user\_id) |

### **12. Student\_Mentor**

| **Column** | **Type** | **Constraints** |
| --- | --- | --- |
| sutdent\_id | INT | PK part, FK → student(user\_id) |
| mentorship\_id | INT | PK part, FK → Alumni(user\_id) |

### 

### 

### 

### **13. User\_Subscriptions**

| **Column** | **Type** | **Constraints** |
| --- | --- | --- |
| user\_id | INT | PK, FK → User(user\_id) |
| subscribed\_events | BOOLEAN | DEFAULT FALSE |
| subscribed\_newsletter | BOOLEAN | DEFAULT FALSE |

### 