Project Specification Document – Group 5 (Design Patterns Class)

Project Group Number/Name

Group 5

Project Topic/Name

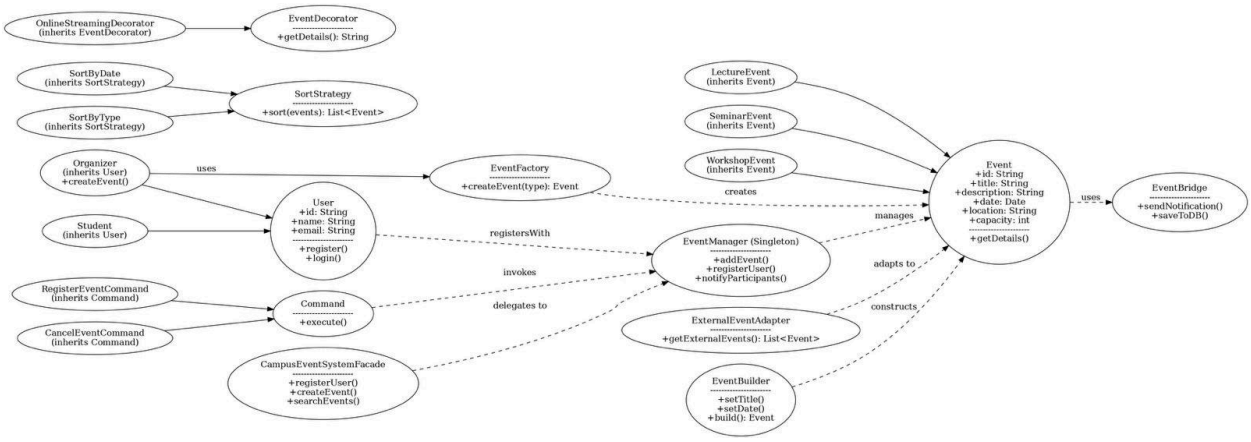
Campus Event Management and Notification System

Problem Statement

University campuses host a variety of events—lectures, workshops, seminars and extracurricular activities—but the information is often scattered across emails, flyers and separate calendars. Students struggle to discover relevant events, organizers cannot easily reach their audience and last‑minute changes lead to confusion. A centralized system is needed where organizers can create different types of events, participants can register or subscribe, and updates propagate automatically. By modelling the domain with object‑oriented principles and leveraging appropriate design patterns, the system should provide a flexible architecture that supports multiple event types, subscription mechanisms and extensibility. The goal is to improve event discovery and attendance, reduce information overload and showcase the application of creational, structural and behavioral design patterns.

UML Diagram/ER Diagram

The following high‑level UML diagram shows the core classes, attributes and relationships. Users (students and organizers) interact with events through an EventManager. Events are created via a factory, participants subscribe to receive notifications, and different algorithms can sort or filter events.



Design Patterns to be Implemented

Design patterns provide reusable solutions to recurring design problems. We plan to implement the following patterns:

Pattern	Purpose in Our System	Reason / Source
Singleton	The EventManager class will be implemented as a Singleton to ensure that only one instance manages all events and users. This pattern ensures a class has only one instance and provides a global point of access	Centralized control of event and user collections simplifies coordination and avoids multiple conflicting managers.
Factory Method	An EventFactory will create concrete event subclasses (LectureEvent, SeminarEvent, WorkshopEvent) based on input parameters. The Factory Method pattern provides an interface for creating objects in a superclass while allowing subclasses to alter the type of objects created	New event types can be added without changing client code, promoting the open/closed principle.
Strategy	Sorting and filtering of events (by date, type or popularity) will use the Strategy pattern. This behavioral pattern defines a family of algorithms, places each in a separate class and makes the algorithms interchangeable	Allows users to choose different sorting/filtering strategies at runtime without modifying the EventManager logic.

Decorator	Optional event features will use the Decorator pattern. Decorators attach new behaviors to objects by wrapping them in special wrapper objects	Enables flexible extension of event functionality without altering existing event classes and avoids subclass explosion.
Facade	Provide a unified and simplified interface (e.g., CampusEventSystemFacade) that coordinates core subsystems like user registration, event creation, searching, and notifications.	Shields clients (UI, API controllers) from internal complexities; centralizes entry points; easier maintenance and testing.
Adapter	Bridge incompatible interfaces to enable integration with external or legacy event feeds/calendars or APIs.	Supports future external integrations without modifying core logic; follows the open/closed principle.
Bridge	Separate event abstractions from implementation details (e.g., notifications, persistence mechanisms).	Increases flexibility, prevents subclass expansion, and enables independent evolution of features.
Command	Encapsulate actions (e.g., event registration, cancellation) as objects for queuing, undo, or audit.	Decouples action requests from execution, supporting features like logging and action history.
Builder	Event creation involves many optional parameters (e.g, with or without speakers, streaming, etc.).	Construct complex event objects step-by-step

Tech Stack

- **Language:** Java 21 (LTS)

- **Build/Version Control:** Maven for dependency management; Git/GitHub for version control and collaboration.
- **Tools:** IDEs such as IntelliJ IDEA or Eclipse; Draw.io for ER diagrams; GitHub Projects.

Functionalities to be Implemented by the End of Milestone 2

By the end of Milestone 2, the group aims to deliver a working prototype that demonstrates the use of the above patterns and provides the following functionality:

- **User management:** basic registration and login for students and organizers. Organizers have privileges to create events, while students can browse and register.

- **Event creation via factory:** organizers can create different types of events (lecture, seminar, workshop) using the EventFactory. Each event has attributes such as title, description, date, location and capacity.

- **Event listing and searching:** users can view an event catalogue, search by keywords and apply sorting/filtering strategies (e.g., sort by date or filter by type) at runtime via the Strategy pattern.

- **Singleton event manager:** a single EventManager instance stores all events and manages registration and notifications. This ensures a consistent state across the application (Singleton pattern).

- **Optional feature decorations:** implement at least one event decorator (e.g., OnlineStreamingDecorator) that can be attached to any event to add additional capabilities such as streaming links or extra fees.

- **Persistence:** Generate output into files.

Contributions of each member:

1. Madhu Sai Kalyan Kaluri - Brainstormed on how to implement Singleton and Factory Singleton design patterns in our project. Will be working on implementing them as a part of the upcoming milestones.

2. Ruthvik Bangalore Ravi - Ideation and implementation of Strategy and Decorator patterns.

3. Khushank Mistry - Will be working on the implementation of the Facade and Adapter patterns.

4. **Mit Sheth** - Will be working on Command and Builder patterns.

5. **Agni Chaturvedi** - Will be working on Bridge pattern and data persistence