Project Specification Document – Group 5 (Design Patterns Class)

Project Group Number/Name

Group 5

Project Topic/Name

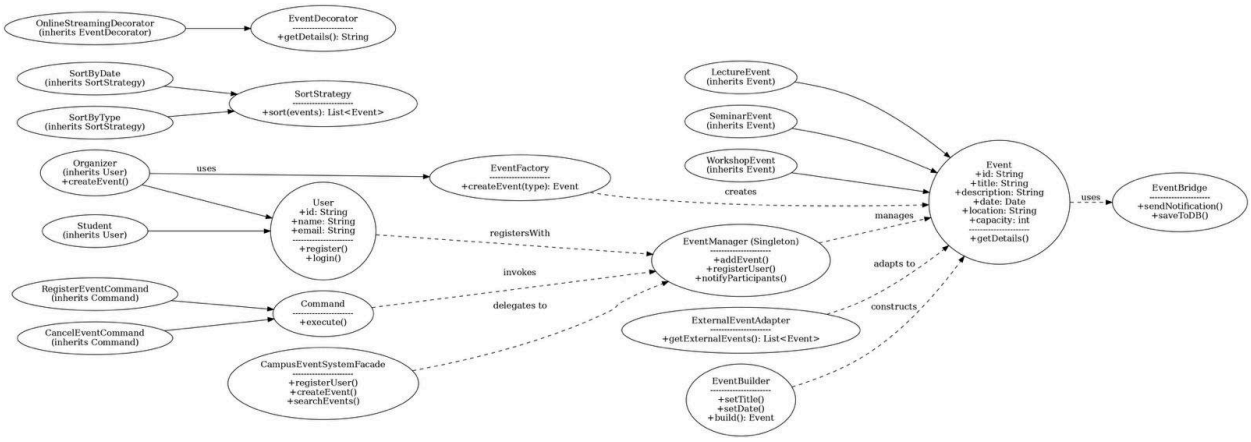
Campus Event Management and Notification System

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 3

By the end of Milestone 3, the group aims to deliver a working prototype that demonstrates the use of the above patterns and provides the following functionality:

- **Command Pattern -** Event Registration/Cancellation System: Implement Command interface with RegisterEventCommand and CancelEventCommand classes to enable undo/redo operations and encapsulate event registration logic with command history tracking and batch operations
- **Decorator Pattern** - Event Enhancement System: Implement EventDecorator abstract class and OnlineStreamingDecorator concrete class to dynamically add features like online streaming, recording functionality, and special access requirements to events without modifying existing event classes
- **Adapter Pattern** - External Event Integration: Implement ExternalEventAdapter to integrate with external systems like Google Calendar, university main calendar, social media platforms, and external notification systems for seamless event synchronization
- **Bridge Pattern** - Event Communication System: Implement EventBridge for different communication channels including email notifications, SMS notifications, push notifications, in-app notifications, and social media announcements to separate event abstraction from communication implementation
- **Facade Pattern** - Campus Event System Facade: Implement CampusEventSystemFacade as a simplified interface providing one-click event creation, simplified event discovery, bulk operations, system-wide search and filtering, and dashboard with key metrics to unify the complex event management system

Contributions of each member:

1. Madhu Sai Kalyan Kaluri - Built base classes for Users, Students and Organizers. Implemented Singleton design pattern (EventManager).

```java
public class EventManager {
    public boolean registerUserForEvent(String userId, String eventId) {
        if (!users.containsKey(userId)) {
            System.out.println("User not found: " + userId);
            return false;
        }

        if (!events.containsKey(eventId)) {
            System.out.println("Event not found: " + eventId);
            return false;
        }

        List<String> registrations = eventRegistrations.get(eventId);
        if (!registrations.contains(userId)) {
            registrations.add(userId);

            // If the user is a student, also update their registered events
            User user = users.get(userId);
            if (user instanceof Student) {
                ((Student) user).registerForEvent(eventId);
            }

            System.out.println("User " + userId + " registered for event " + eventId);
            return true;
        } else {
            System.out.println("User " + userId + " is already registered for event " + eventId);
            return false;
        }
    }

    public void notifyParticipants(String eventId, String message) {
        if (!events.containsKey(eventId)) {
            System.out.println("Event not found: " + eventId);
            return;
        }
```

```java
public class EventManager {

    private static EventManager instance;

    private Map<String, Event> events;
    private Map<String, User> users;
    private Map<String, List<String>> eventRegistrations;

    private EventManager() {
        events = new HashMap<>();
        users = new HashMap<>();
        eventRegistrations = new HashMap<>();
    }

    public static EventManager getInstance() {
        if (instance == null) {
            instance = new EventManager();
        }
        return instance;
    }

    public boolean addEvent(Event event) {
        if (event != null && event.getId() != null) {
            events.put(event.getId(), event);
            eventRegistrations.put(event.getId(), new ArrayList<>());
            System.out.println("Event added: " + event.getTitle() + " (ID: " + event.getId() + ")");
            return true;
        }
        return false;
    }

    public boolean registerUser(User user) {
        if (user != null && user.getId() != null) {
            users.put(user.getId(), user);
            System.out.println("User registered: " + user.getName() + " (ID: " + user.getId() + ")");
```

2. Ruthvik Bangalore Ravi - Implemented Events model class with factory, added strategy pattern to sort events by title and date

```java
public class LectureEventFactory extends AbstractEventFactory {  2 usages  Ruthvikbr
    @Override  1 usage  Ruthvikbr
    public Event createEvent(String id, String title, String description, LocalDate date, String location, int capacity) {
        return new LectureEvent(id, title, description, date, location, capacity);
    }
}
```

```java
public class SortByTitle implements SortStrategy {  3 usages  Ruthvikbr
    @Override  2 usages  Ruthvikbr
    public List<Event> sortEvents(List<Event> events) {
        System.out.println("Sorting events by title...");

        events.sort(Comparator.comparing(Event::getTitle));
        return events;
    }
}
```

3. Khushank Mistry - Added Facade Design Pattern and Search and Notification service.

```java
public class CampusEventSystemFacade {

    private static CampusEventSystemFacade instance;

    // Core subsystems
    private final EventManager eventManager;
    private final NotificationService notificationService;
    private final SearchService searchService;

    // Event factories
    private final Map<String, AbstractEventFactory> eventFactories;

    // External adapters
    private final List<ExternalEventAdapter> externalAdapters;

    private CampusEventSystemFacade() {
        // Initialize core subsystems
        this.eventManager = EventManager.getInstance();
        this.notificationService = new NotificationService();
        this.searchService = new SearchService();

        // Initialize event factories
        this.eventFactories = new HashMap<>();
        this.eventFactories.put(key:"lecture", new LectureEventFactory());
        this.eventFactories.put(key:"seminar", new SeminarEventFactory());
        this.eventFactories.put(key:"workshop", new WorkshopEventFactory());

        // Initialize external adapters list
        this.externalAdapters = new ArrayList<>();
    }

    public static CampusEventSystemFacade getInstance() {
        if (instance == null) {
            instance = new CampusEventSystemFacade();
        }
```

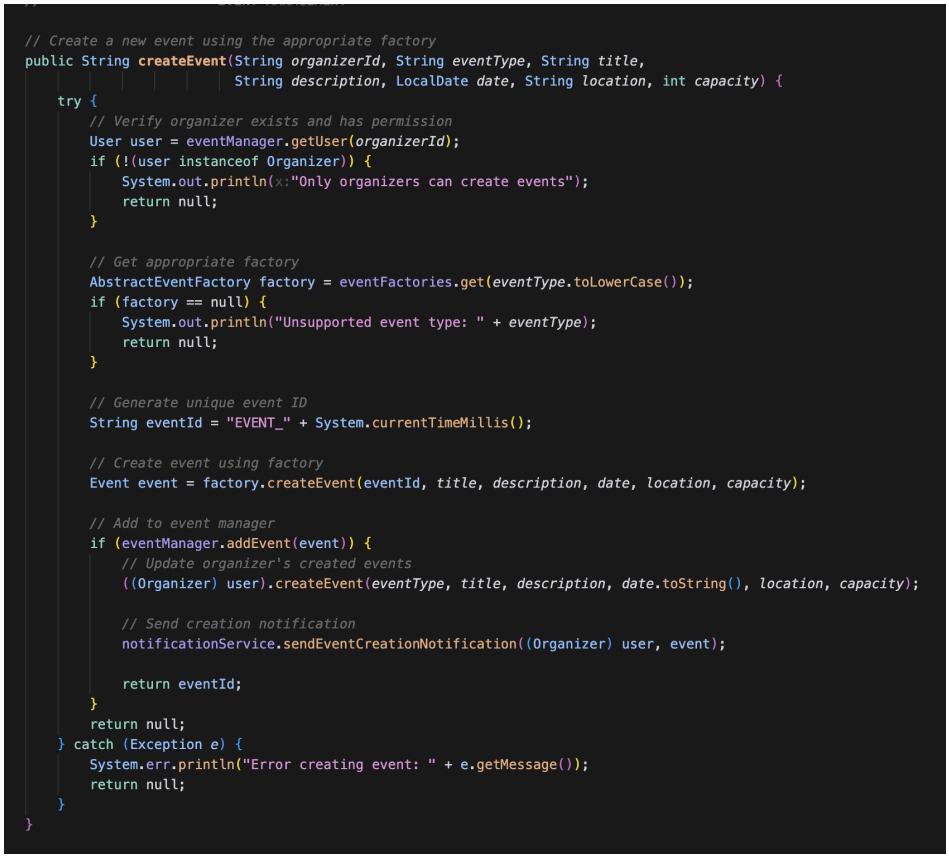
```java
// Register a new student in the system

public boolean registerStudent(String id, String name, String email,
                               String studentId, String major, int year) {
    try {
        Student student = new Student(id, name, email, studentId, major, year);
        boolean registered = student.register();
        if (registered) {
            eventManager.registerUser(student);
            notificationService.sendWelcomeNotification(student);
            return true;
        }
        return false;
    } catch (Exception e) {
        System.err.println("Error registering student: " + e.getMessage());
        return false;
    }
}


// Register a new organizer in the system
public boolean registerOrganizer(String id, String name, String email,
                                 String department, String role) {
    try {
        Organizer organizer = new Organizer(id, name, email, department, role);
        boolean registered = organizer.register();
        if (registered) {
            eventManager.registerUser(organizer);
            notificationService.sendWelcomeNotification(organizer);
            return true;
        }
        return false;
    } catch (Exception e) {
        System.err.println("Error registering organizer: " + e.getMessage());
```

```java
public boolean registerOrganizer(String id, String name, String email,
        Organizer organizer = new Organizer(id, name, email, department, role);
        boolean registered = organizer.register();
        if (registered) {
            eventManager.registerUser(organizer);
            notificationService.sendWelcomeNotification(organizer);
            return true;
        }
        return false;
    } catch (Exception e) {
        System.err.println("Error registering organizer: " + e.getMessage());
        return false;
    }
}


// Authenticate user login
public boolean loginUser(String userId) {
    try {
        User user = eventManager.getUser(userId);
        if (user != null) {
            boolean loginSuccess = user.login();
            if (loginSuccess) {
                notificationService.sendLoginNotification(user);
            }
            return loginSuccess;
        }
        System.out.println("User not found: " + userId);
        return false;
    } catch (Exception e) {
        System.err.println("Error during user login: " + e.getMessage());
        return false;
    }
}
```



```java
// Create a new event using the appropriate factory
public String createEvent(String organizerId, String eventType, String title,
                          String description, LocalDate date, String location, int capacity) {
    try {
        // Verify organizer exists and has permission
        User user = eventManager.getUser(organizerId);
        if (!(user instanceof Organizer)) {
            System.out.println("Only organizers can create events");
            return null;
        }

        // Get appropriate factory
        AbstractEventFactory factory = eventFactories.get(eventType.toLowerCase());
        if (factory == null) {
            System.out.println("Unsupported event type: " + eventType);
            return null;
        }

        // Generate unique event ID
        String eventId = "EVENT_" + System.currentTimeMillis();

        // Create event using factory
        Event event = factory.createEvent(eventId, title, description, date, location, capacity);

        // Add to event manager
        if (eventManager.addEvent(event)) {
            // Update organizer's created events
            ((Organizer) user).createEvent(eventType, title, description, date.toString(), location, capacity);

            // Send creation notification
            notificationService.sendEventCreationNotification((Organizer) user, event);

            return eventId;
        }
        return null;
    } catch (Exception e) {
        System.err.println("Error creating event: " + e.getMessage());
        return null;
    }
}
```

4. **Mit Sheth** - Implemented the Builder Design Pattern (EventBuilder.java) to support flexible and fluent creation of complex Event objects, including optional parameters and support for subtypes (LectureEvent, SeminarEvent, WorkshopEvent).

EventBuilder.java | LectureEvent.java | WorkshopEvent.java | SeminarEvent.java

src > main > java > edu > neu > csye7374 > event > EventBuilder.java

```java
package edu.neu.csye7374.event;

import java.time.LocalDate;
import java.util.UUID;

public class EventBuilder {
    private String type = "lecture";  // default
    private String title;
    private String description;
    private LocalDate date;
    private String location;
    private int capacity;

    public EventBuilder setType(String type) {
        this.type = type.toLowerCase();
        return this;
    }

    public EventBuilder setTitle(String title) {
        this.title = title;
        return this;
    }

    public EventBuilder setDescription(String description) {
        this.description = description;
        return this;
    }

    public EventBuilder setDate(LocalDate date) {
        this.date = date;
        return this;
    }

    public EventBuilder setLocation(String location) {
        this.location = location;
        return this;
    }

    public EventBuilder setCapacity(int capacity) {
        this.capacity = capacity;
        return this;
    }

public Event build() {
    String id = UUID.randomUUID().toString();

    switch (type) {
        case "seminar":
            return new SeminarEvent(id, title, description, date, location, capacity);
        case "workshop":
            return new WorkshopEvent(id, title, description, date, location, capacity);
        default:
            return new LectureEvent(id, title, description, date, location, capacity);
    }
}
}

```

5. **Agni Chaturvedi -** Built Bridge design pattern to publish Events.

```java
package edu.neu.csye7374.bridge;

public abstract class BridgeEvent {
    protected EventPublisher publisher;

    public BridgeEvent(EventPublisher publisher) {
        this.publisher = publisher;
    }

    public abstract void publish();
}
```

```java
package edu.neu.csye7374.bridge;

import edu.neu.csye7374.event.Event;

public interface EventPublisher {
    void publish(Event event);
}
```

```java
package edu.neu.csye7374.bridge;

import edu.neu.csye7374.event.Event;

public class ConsoleEventPublisher implements EventPublisher {
    @Override
    public void publish(Event event) {
        System.out.println("Publishing event to console: " + event);
    }
}
```

```java
package edu.neu.csye7374.bridge;

import edu.neu.csye7374.event.Event;

import java.io.FileWriter;
import java.io.IOException;

public class FileEventPublisher implements EventPublisher {
    @Override
    public void publish(Event event) {
        try (FileWriter fw = new FileWriter(fileName:"events_output.txt", append:true)) {
            fw.write(event.toString() + "\n");
            System.out.println(x:"Event written to file.");
        } catch (IOException e) {
            System.err.println("Error writing event to file: " + e.getMessage());
        }
    }
}
```