UE21CS352B - Object Oriented Analysis & Design using Java

Mini Project Report

PESU Clubs Management

Submitted by:

Sana Suman	**PES1UG21CS911**
Mutasim	**PES1UG21CS338**
Manoj Kumar H S	**PES1UG21CS329**
M S Shriya	**PES1UG21CS331**

6th Semester F Section

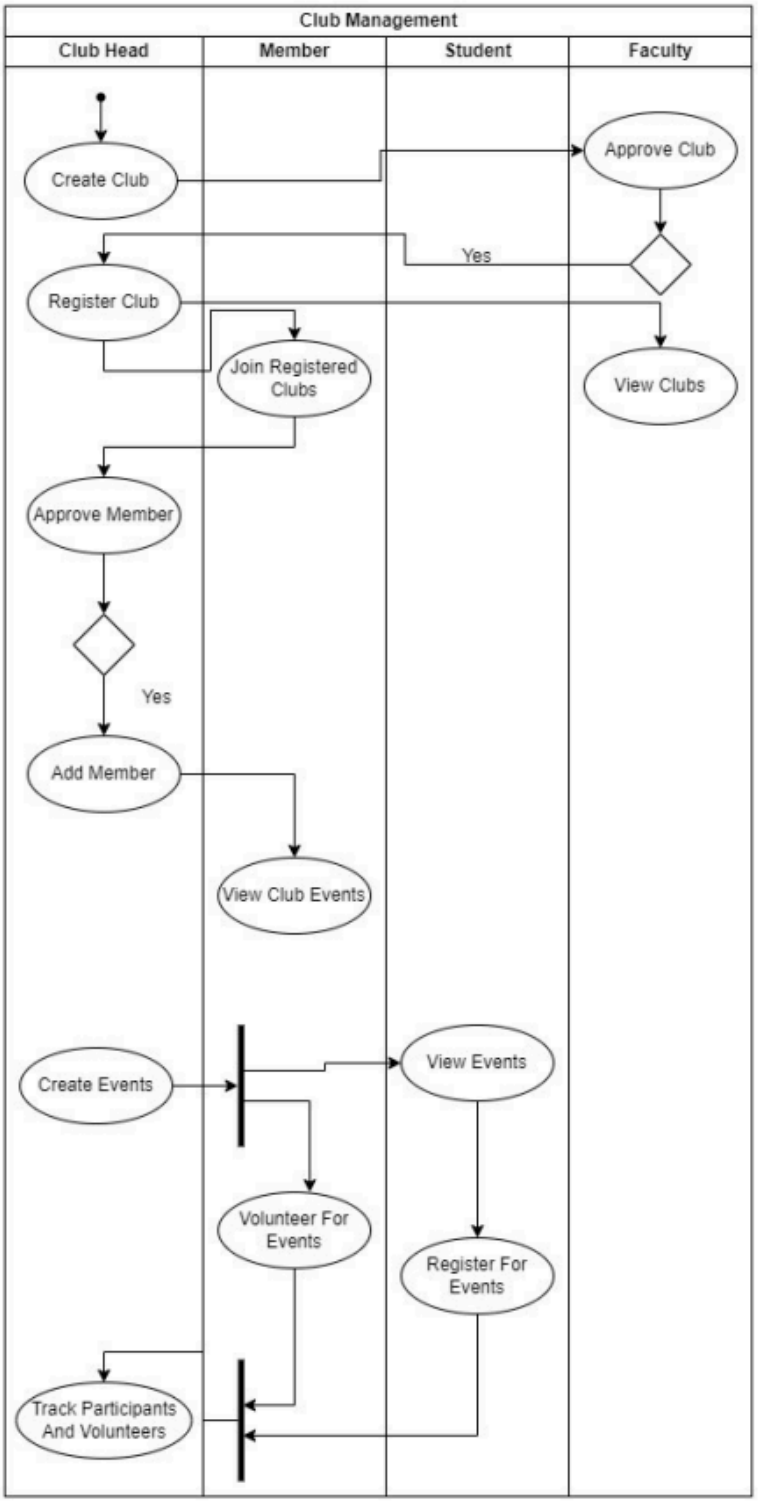
Prof. Bhargavi Mokashi
Assistant Professor, Dept. of CSE

January - May 2024

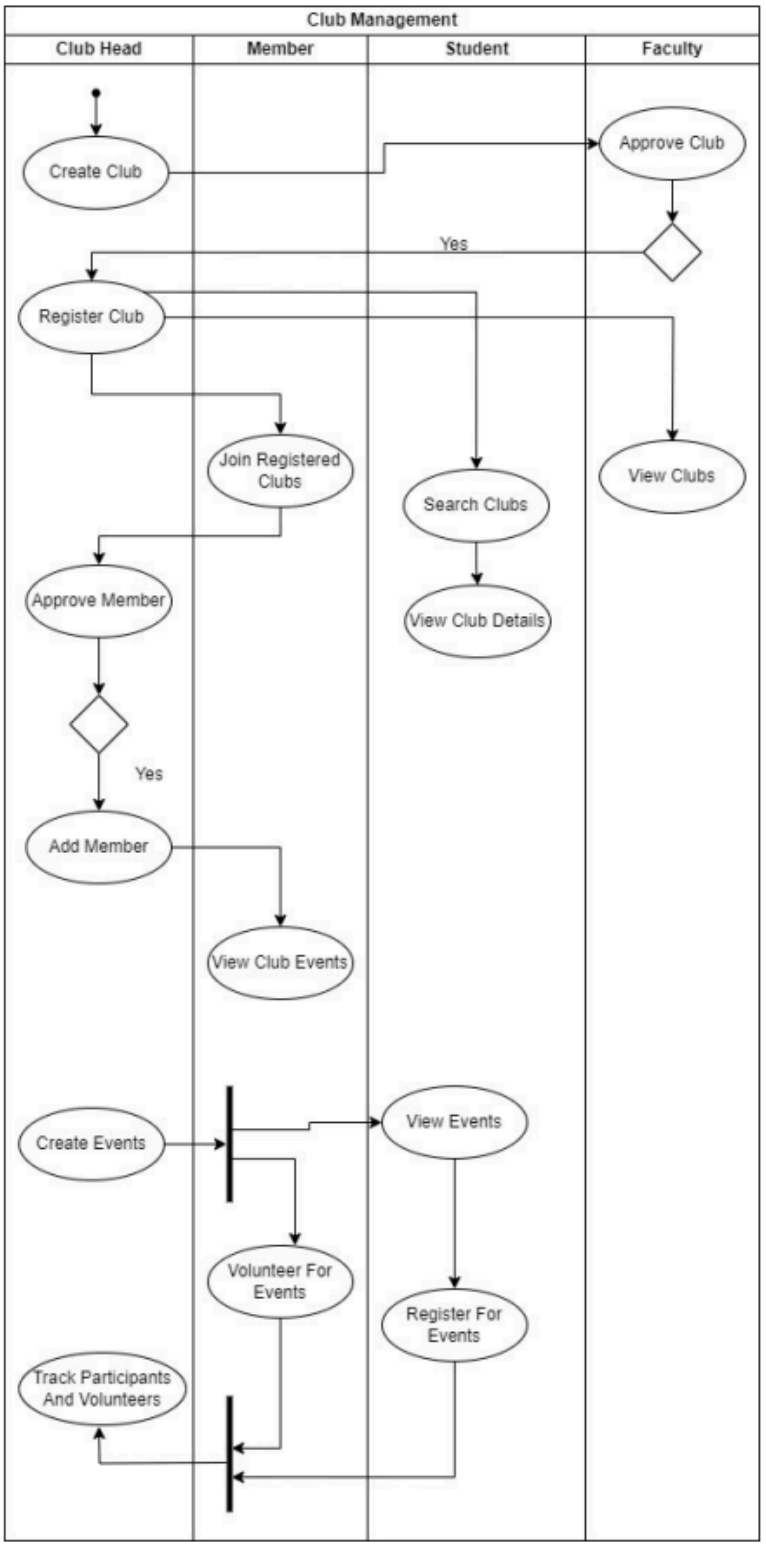
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

Table of Contents

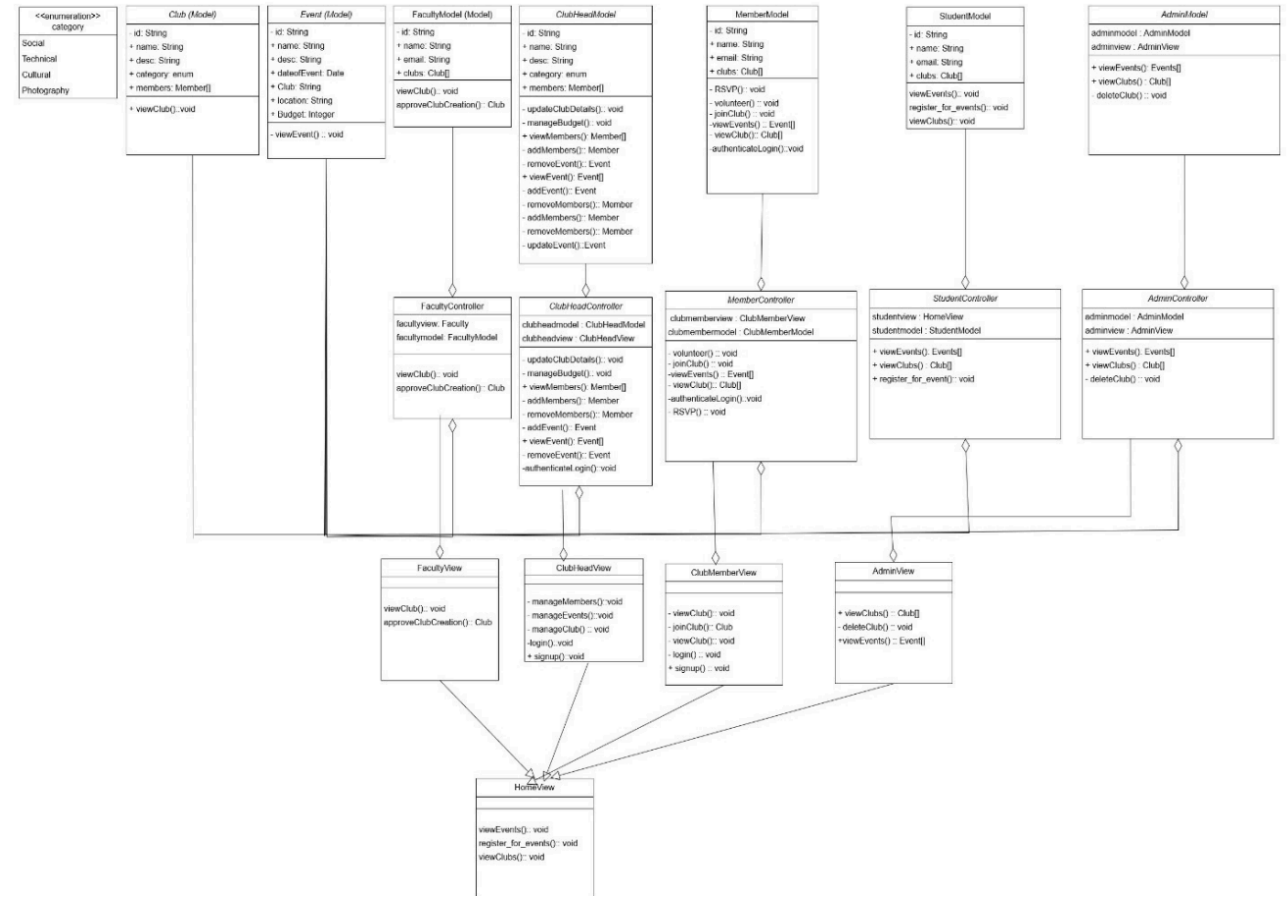
Generalized Activity Diagram:



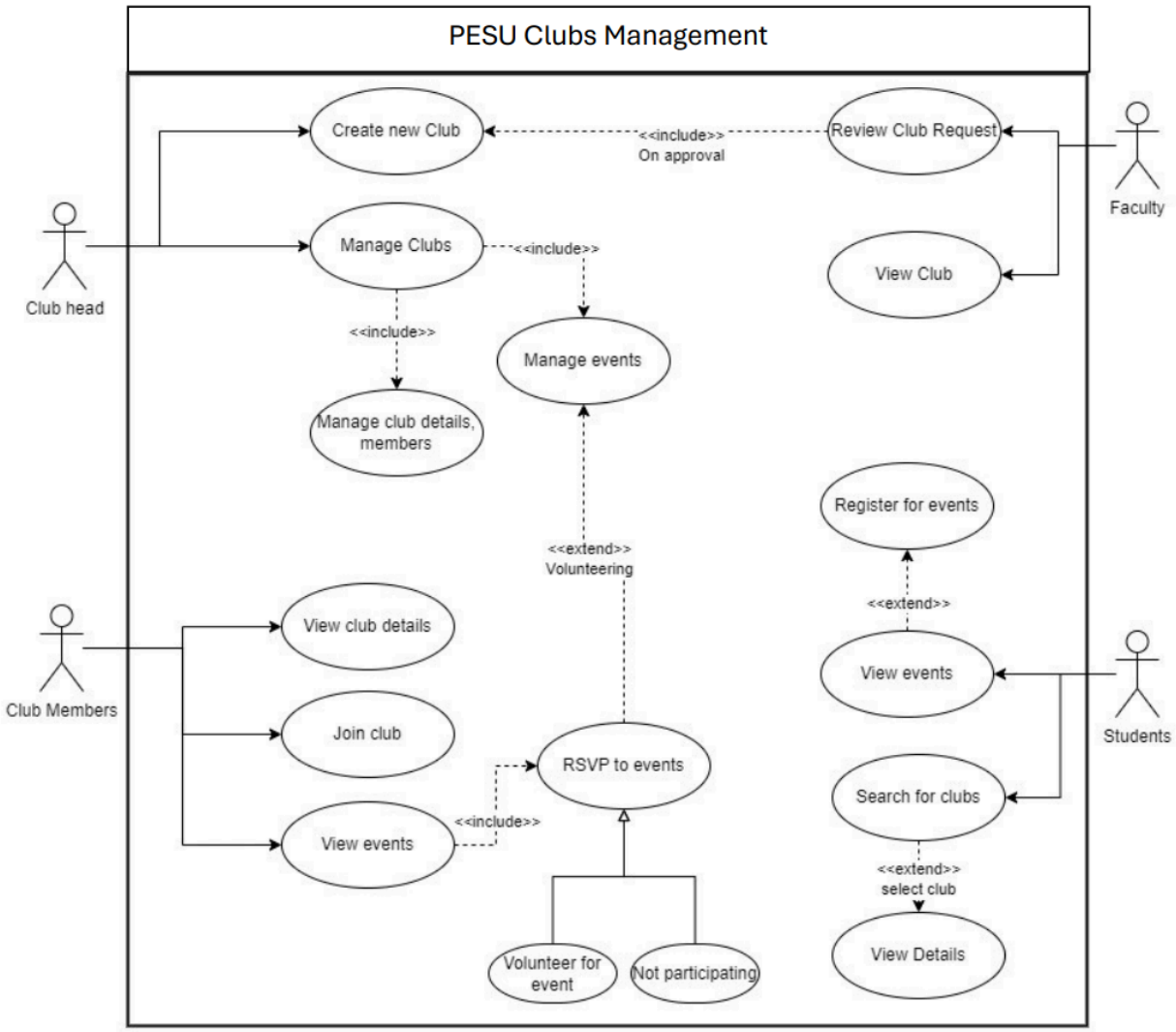
Activity Diagram for Specific Use Case : Event Management



Class Diagram:

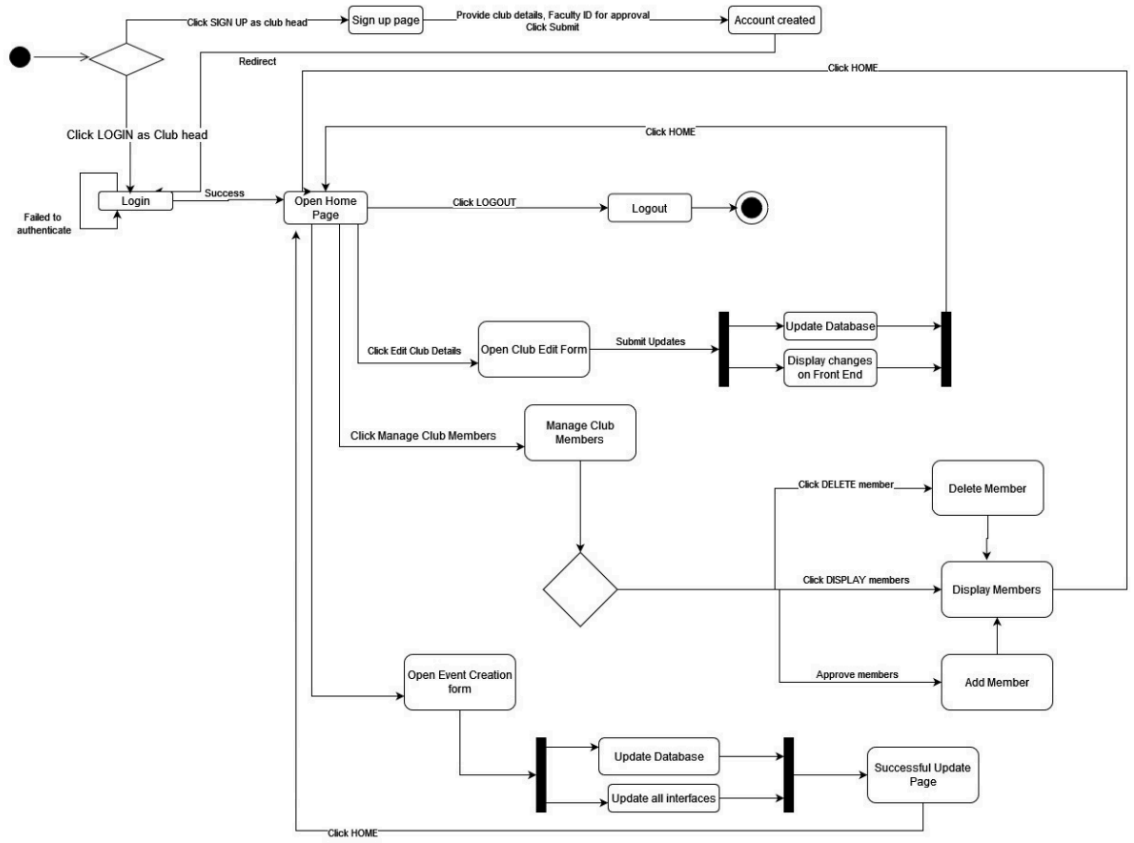


Use Case Diagram:

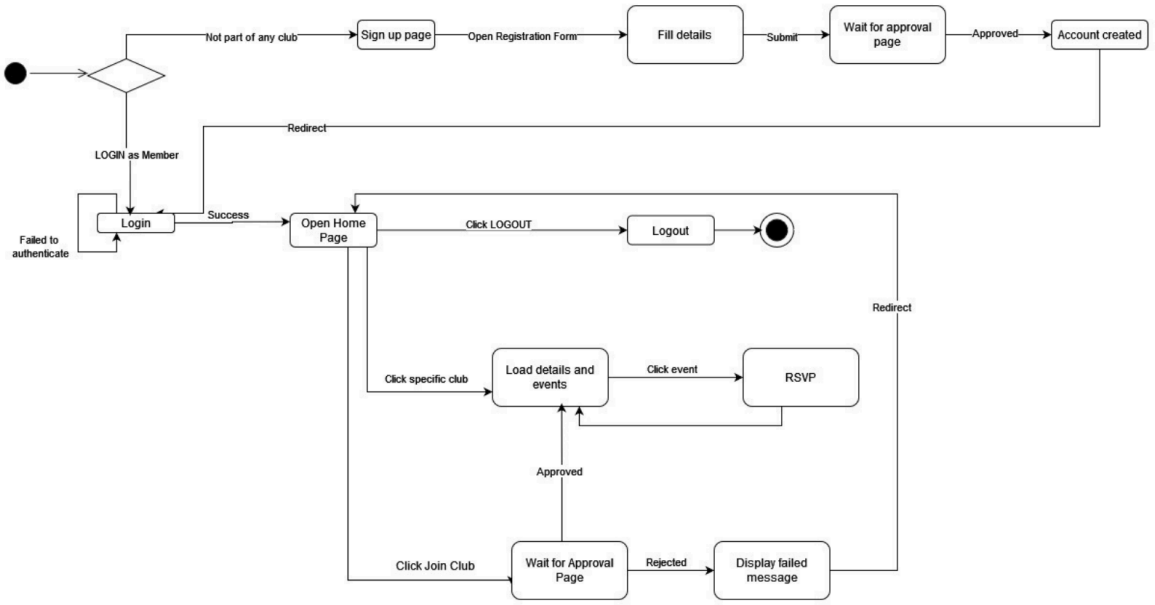


State Diagram:

For Club Admin:



For Club Member:



Patterns

Factory Pattern - Creational

The Factory pattern is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created.

The Factory pattern is applied in our project to encapsulate the object creation process within dedicated factory classes **MemberFactory** and **ClubFactory**. These factories centralize the creation logic for their respective objects **Member** and **Club**. The services, **MemberService** and **ClubService** then use these factories to create instances of the respective domain objects, keeping the creation logic separate from other parts of the application.

This approach promotes encapsulation, flexibility, and easier maintenance by isolating the object creation process from the rest of the application logic.

Proxy Pattern - Structural

The Proxy pattern is a structural design pattern that provides a placeholder for another object to control access to it. The proxy object acts as an intermediary, allowing clients to interact with the real object indirectly.

In our project, the Proxy pattern is implemented with the **EventServiceProxy** class acting as a proxy for the **EventService class.** The **EventServiceInterface** defines methods for managing events, including **getAllEvents()** to retrieve all events and **createEvent(Event event)** to create a new event. The **EventService** implements this interface, handling event operations by utilizing repository methods like **eventRepository.findAll()** and **eventRepository.save(event)**. Acting as a proxy, the **EventServiceProxy** intercepts calls to the service methods, executing additional logic before or after delegation to the actual **EventService** instance, **eventService.getAllEvents()** and **eventService.createEvent(event)**. This pattern allows for access control, logging, or caching functionalities to be seamlessly integrated into event management operations in the following manner.

Principles

Single Responsibility Principle (SRP)

By having separate controllers for club, member, home page, authentication, and their respective models, repositories, and services, we ensure SRP. Each class or component has a single, well-defined responsibility, promoting code maintainability and extensibility.

Controller

The application follows the Controller principle by having dedicated controller classes that handle incoming requests, process user input, and coordinate the flow of the application.

Information Expert

The ClubService, MemberService, EventService etc act as Information Experts, encapsulating the business logic and rules related to specific entities or domains, such as club management, member management, or authentication.

Low Coupling

By separating concerns into different layers (controllers, services, repositories, models), the application promotes low coupling between components, making it easier to maintain, test, and extend the codebase.

High Cohesion

By separating the responsibilities and functionalities into different components like MemberDashboard and Club, we ensure that each component has a clear and well-defined purpose. This high cohesion makes it easier to understand, maintain, and modify the codebase, as changes to a specific concern are localized within the corresponding component.