**Reflection on Design Patterns Chosen**

In the design of the **Community Event Web Application**, the use of appropriate software design patterns was a critical step in achieving a modular, scalable, and maintainable system. Each pattern chosen addressed a specific design problem and contributed to the overall efficiency and flexibility of the application architecture.

**1. Singleton Pattern**

The **Singleton Pattern** was implemented for the **NotificationService** component to ensure that only a single instance of the notification handler exists throughout the system. This decision was driven by the need to maintain consistency in managing real-time notifications for users when new events are created or updates occur. By restricting the class to one instance, it prevents resource conflicts and simplifies coordination between the backend and notification channels. This pattern enhanced efficiency and reduced redundancy, especially in managing multiple concurrent notification requests.

**2. Factory Pattern**

The **Factory Pattern** was chosen to handle the creation of different types of notifications such as **email alerts**, **push notifications**, or **SMS updates**. Instead of tightly coupling the notification logic to one specific implementation, the Factory Pattern allows the system to dynamically generate the appropriate notification object based on context. This promotes scalability and flexibility, especially as future versions of the application may integrate additional communication methods. Through this approach, the system remains open for extension but closed for modification—following the **Open/Closed Principle** of object-oriented design.

**3. Observer Pattern**

The **Observer Pattern** was applied between the **Event** and **Notification** components to facilitate **real-time updates**. Whenever a user registers for an event or when an event is updated, all subscribed users (observers) are automatically notified. This event-driven design supports the application’s instant notification feature and strengthens the interaction between system modules. The pattern helped achieve a decoupled communication structure, making it easier to modify or extend event behavior without altering the notification logic.

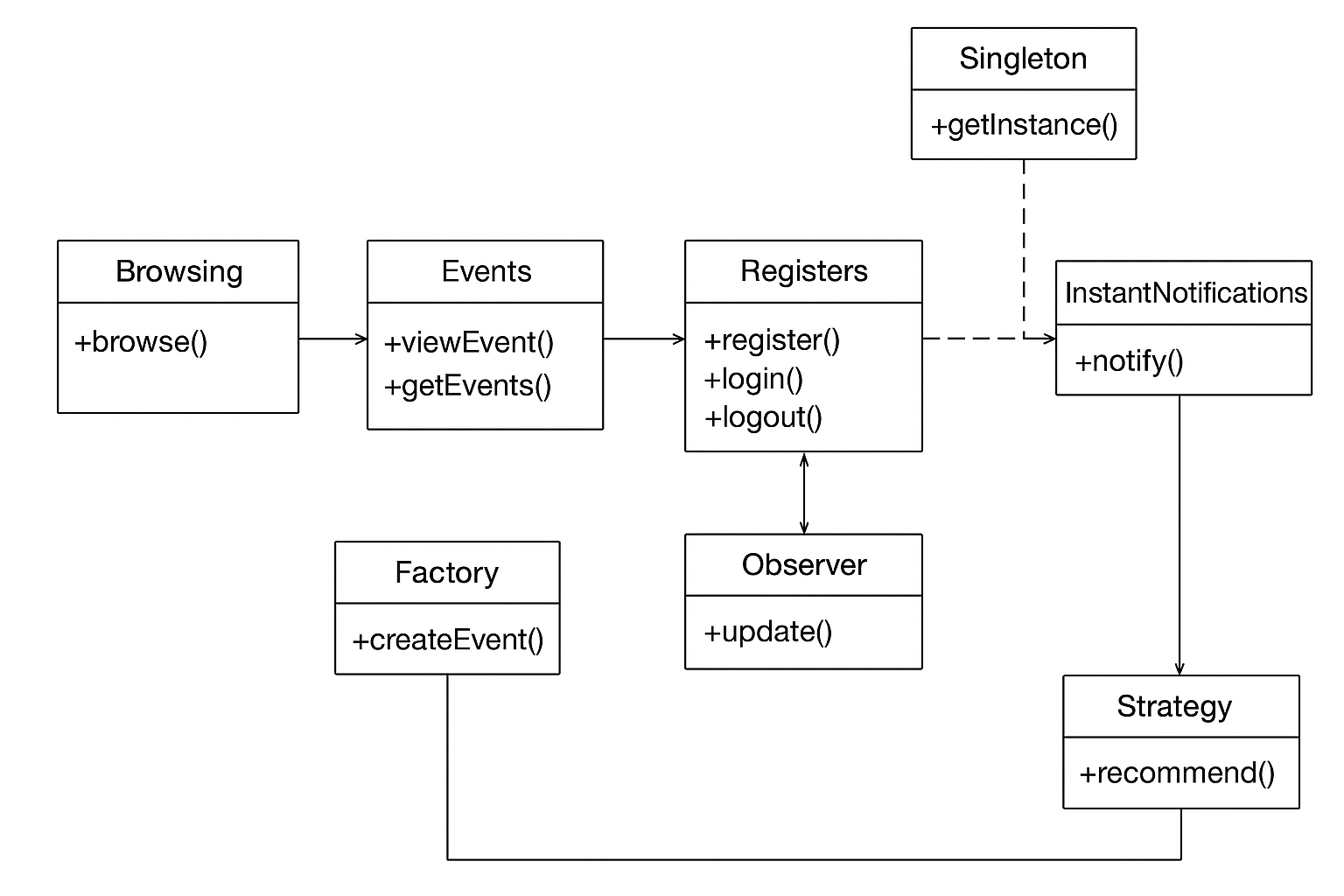
**4. Strategy Pattern**

The **Strategy Pattern** was integrated into the **AI Event Suggester** module. Since multiple algorithms can be used to recommend events—such as filtering by category, popularity, or user similarity—the Strategy Pattern allows these algorithms to be interchanged easily at runtime. This flexibility supports experimentation and optimization of AI models without affecting the rest of the system. It also improves code maintainability and supports continuous improvement of the recommendation logic.

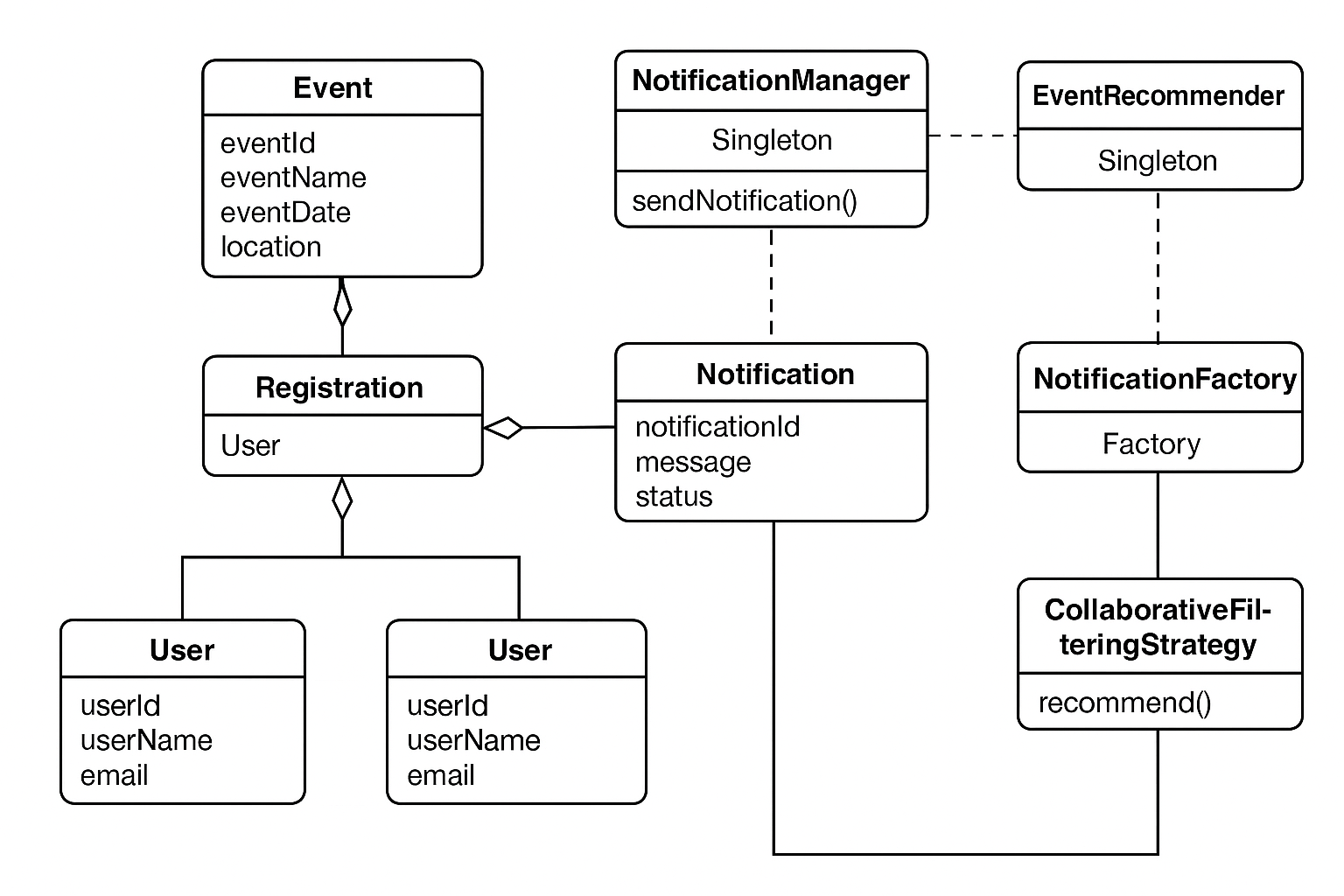
## ****Project Reflection: Community Event Web Application****

The development of the **Community Event Web Application** provided an opportunity to transform a real-world problem into a functional, technology-driven solution. The main goal was to create a platform that allows community members to **browse and register for events**, receive **instant notifications**, and benefit from an **AI-powered event suggester** that personalizes recommendations based on user interests and activity history.

Designing UML diagrams such as **Class and Sequence Diagrams** helped visualize both the **static and dynamic structure** of the system. These visual models provided clarity on how system components interact, how data flows between modules, and how design patterns (such as **Singleton, Factory, Observer, and Strategy**) ensure reusability and maintainability.



Sequence Diagram

 Class Diagram

Working on this proposal also strengthened my understanding of **software design principles**, **system architecture**, and **real-time communication technologies**. I realized that clear planning and visual modeling are crucial before implementation—helping developers, stakeholders, and designers align on the same vision.

Overall, this project proposal has not only expanded my technical skills but also deepened my appreciation for systematic software engineering. It taught me the importance of **design thinking**, **user-centered development**, and **continuous learning** in building applications that are practical, intelligent, and socially beneficial.

**Reflection Summary**

In conclusion, the combination of **Singleton**, **Factory**, **Observer**, and **Strategy** patterns created a balanced and flexible architecture. Each pattern addressed a distinct design challenge—ranging from instance management and object creation to communication and algorithm selection. Applying these patterns not only improved **system reusability** and **scalability** but also deepened my understanding of how design principles translate into practical software engineering solutions. Overall, these design patterns collectively strengthened the application’s structure, maintainability, and ability to adapt to future technological changes.