One of the non-functional requirements is about the system providing events based on the personal preferences of the user. To satisfy this requirement we tried to use the following patterns

Design Patterns Used in the System

1. Bridge Pattern

Definition:
The Bridge Pattern is a structural design pattern that decouples an abstraction from its implementation so they can evolve independently.

Application in Our System:
We use the bridge to separate the **display logic** (e.g., `EventTabGUI`, `EventRecommendationFacade`) from the **recommendation logic** (`RecommendationStrategy` and its implementations).

2. Composite Pattern

Definition:
The Composite Pattern treats individual objects and compositions of objects uniformly.

Application in Our System:
`CompositeRecommendationStrategy` groups multiple recommendation strategies and treats them as a single unit.

3. Strategy Pattern

Definition:
The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable.

Application in Our System:
Each recommendation criterion (e.g., likes, profile interests, proximity) is implemented as a separate strategy class that follows a common interface.

4. Proxy Pattern

Definition:
The Proxy Pattern provides a surrogate or placeholder to control access to another object.

Application in Our System:
`LocationProxy` manages access to the user's location services.

Why We Chose It:

- Location data is sensitive and must be accessed securely.
- The proxy checks for permissions, handles retries, and manages timeouts.
- Ensures compliance with privacy regulations (e.g., GDPR) without polluting the core logic.

5. Adapter Pattern

Definition:
The Adapter Pattern allows objects with incompatible interfaces to work together.

Application in Our System:
`LocationServiceAdapter` adapts the Google Maps API for use within our internal architecture.

Why We Chose It:

- To isolate external APIs from internal business logic.
- To support switching map providers with minimal code change.
- To standardize location data handling across the system.

Recommendation Flow and Class Responsibilities

Below all the methods are explained.

1. `eventTabGUI`

Methods

- `displayEvents(events)` – Displays personalized events to the user.
- `showError(message)` – Shows an error message to the user.

Connected To

- **`EventRecommendationFacade`** — Acts as the abstraction in the **Bridge Pattern**. It delegates recommendation logic and stays independent from implementation.

2. `EventRecommendationFacade`

Methods

- `recommendEvents(user)` – Produces a list of recommended events.
- `setUserPreferences(user)` – Sets user preferences for future recommendations.
- `logError(message)` – Logs error messages internally.
- `buildRecommendationResponse()` – Constructs a standardized response object.

Uses

- **`RecommendationStrategy`** — Applied via the **Strategy Pattern**, enabling plug-and-play recommendation logic.
- Encapsulates multiple strategies using **`CompositeRecommendationStrategy`** — following the **Composite Pattern**.

3. `CompositeRecommendationStrategy`

Methods

- `recommend(user)` – Aggregates results from all internal strategies.
- `addStrategy(strategy)` – Adds a strategy dynamically.
- `removeStrategy(strategy)` – Removes a strategy.
- `clearStrategies()` – Clears all strategies.
- `aggregateResults()` – Combines recommendation outputs.

Includes

- `LikesStrategy`
- `ProfileStrategy`
- `LocationStrategy`

> Implements the **Composite Pattern** to group strategies and treat them uniformly.

4. `LikesStrategy`

Methods

- `recommend(user)` – Suggests events based on user's liked events.

> Implements **Strategy Pattern**: one concrete strategy for personalization.

5. `ProfileStrategy`

Methods

- `recommend(user)` – Suggests events based on user's profile interests.

> Another implementation of the **Strategy Pattern**.

6. `LocationStrategy`

Methods

- `recommend(user)` – Suggests events based on proximity.
- `setLocationService(service)` – Sets the service used to retrieve location.
- `filterEventsByDistance(events)` – Filters events by radius/distance.

Uses

- **`LocationServiceAdapter`** — Uses **Adapter Pattern** to communicate with external location APIs.

7. `LocationServiceAdapter`

Methods

- `getCoordinates()` – Retrieves user's coordinates.
- `validateAccess(user)` – Checks if location access is permitted.

Uses

- **`LocationProxy`** — via **Proxy Pattern** to securely access user location.

8. `LocationProxy`

Methods

- `getCoordinates()` – Retrieves coordinates safely.
- `retryRequest()` – Retries location request on failure.
- `checkAccess(user)` – Ensures user permission exists.
- `handleTimeout()` – Handles request timeout scenarios.

Uses

- **`GoogleMapsService`** — Actual provider of location data.

9. `GoogleMapsService`

Methods

- `getCoordinates()` – Gets coordinates from Google Maps API.
- `fetchCoordinatesFromAPI()` – Sends API request.
- `handleExternalError()` – Manages API-level errors.

> Connected through **Adapter + Proxy** to isolate and secure API interaction.