Project: Event Processing System – Original Code Review
Design Decision Document

Hamza Zarour

1) Why each design pattern was chosen

Observer Pattern (for notifications)

In the original design, EventProcessor directly called Dashboard and Logger inside the process() method. This made the processor tightly coupled with these modules.

The Observer pattern was chosen to:

- Separate notification logic from the core event processing logic.
- Allow Dashboard and Logger to react to events without the processor knowing their details.
- Make it easy to add new notification modules in the future without modifying EventProcessor.

Decorator Pattern (for data transformation)

Originally, the payload transformation logic (encrypt, compress, add metadata) was implemented using multiple if statements inside EventProcessor.

The Decorator pattern was chosen to:

- Remove transformation if statements from the processor.
- Represent each transformation as a separate class.
- Allow flexible combination of transformations based on event flags.
- Keep the prosessor clean and focused only on orchestration.

Strategy Pattern (for event type behavior)

The original code used a large if/else block based on event type strings ("USER", "SYSTEM", "SECURITY") inside EventProcessor.

The Strategy pattern was chosen to:

- Move the behavior of each event type into its own class.
- Avoid a growing if/else block in the processor.
- Make it easy to add new event types by creating new strategies instead of modifying the prosessor logic.

2) What alternatives you considered

Observer Pattern

The alternative was to keep direct calls to Dashboard and Logger inside EventProcessor.
This approach was simpler but caused tight coupling and make extension difficult.
Observer provides a cleaner and more flexible solution.

Decorator Pattern

I considered keeping transformation logic using if statements or using one large transformer class.

These options would grow over time and make the processor harder to maintain.

Decorator is more suitable for optional and combinable behaviors.

Strategy Pattern

The alternative was to keep the existing if/else or replace it with a switch statement.

Both approaches still centralize logic in one class and violate OCP.

Strategy allows each behavior to evolve independently.

3) How the patterns interact

The patterns work together in a clear processing pipeline:

1. **Decorator**
   - TransformerChainBuilder builds a transformation chain based on event flags.
   - The payload is processed using decorators before saving.
2. **Observer**
   - After the event is saved, EventPublisher notifies all registered listeners.

- DashboardListener and LoggerListener receive the processed event.

3. **Strategy**
    - After notification, EventStrategyFactory selects the correct strategy.
    - The selected strategy executes type-specific behavior for the event.

Each pattern has a clear role and does not interfere with the others

4) How the architecture improves scalability, flexibility, and maintainability

Scalability

- New event types can be added without modifying EventProcessor.
- New transformations can be added without changing existing transformation logic.
- New observars can be added without touching the processor.

Flexibility

- Event behavior, transformations, and notifications can evolve independently.
- The system can be extended by adding new classes rather than modifying core logic.

Maintainability

- Responsibilities are clearly separeted:

    EventProcessor coordinates the flow.

    Decorators handle payload transformation.

    Strategies handle type-specific behavior.

    Observers handle notifications.

- Classes are smaller, easier to read, and easier to test