Issue 1 – The original process(Event e) did “everything”:

The old `process()` method was huge. It handled validation, notifying other components, transforming the payload, saving to the database, and also the type-specific actions all in one place.

This breaks the SRP. One method had too many jobs, which makes the code harder to read, test, and safely change.

After Refactor:

I divided the work into smaller, focused parts:

1. Validation moved to `isValid()`
2. Notifications handled through `notifyObservers()`
3. Payload transformation moved to `EventTransformer`
4. Type-specific behaviour moved into separate strategy classes

Patterns I used: Strategy, Template Method, Observer.

Issue 2 – Type checking with String if/else blocks:

The old code checked the event type using Strings like `"USER"`, `"SYSTEM"`, `"SECURITY"`.

These are magic strings and easy to mistype. Every new event type means editing the same long method, which breaks the OCP.

After Refactor:
We replaced type Strings with an `EventType` enum and connected each type to its own strategy in a map.

Patterns I used: Strategy, Enum.

Issue 3 – Direct dependency on concrete classes:

`EventProcessor` used the concrete classes `Database`, `Dashboard`, and `Logger` directly, this creates tight coupling. It becomes harder to replace, extend, or mock these components for testing.

After Refactor:

1. Added a `Database` interface + `DatabaseProxy` to control access
2. Added an `EventObserver` interface
3. Converted `Dashboard` and `Logger` into observers instead of hard-wired calls

Patterns used: Proxy, Observer, Dependency Inversion.

Issue 4 – No connection pooling:

Database access was very basic — no reuse of connections or realistic resource handling.
Not scalable, not efficient, and not close to how real systems manage DB resources.

After Refactor:

I add:

1. `DbConnectionPool` to manage reusable connections
2. `DbConnection` representing one connection
3. `DatabaseProxy` to hide the pooling and simplify saving

Patterns I used: Connection Pool, Proxy.

Issue 5 – Transformation logic mixed inside EventProcessor:

All transformation steps (encrypt, compress, metadata) were implemented inside the processor using flags and manual string editing, this breaks SRP, mixes concerns, and makes it hard to test or add new transformation rules later.

After Refactor:

1. Moved the main transformation workflow into `EventTransformer.transform()`
2. Each feature became a decorator: `EncryptDecorator`, `CompressDecorator`, `MetadataDecorator`

Patterns I used: Template Method, Decorator.

Issue 6 – Changing the original Event object directly:

The processor edited the same `Event` object during processing (like updating the ID),It makes the event state unclear and can affect code that still references the original event.

After Refactor:
Added a `copy()` function and worked on a clone (`workingCopy`) instead of modifying the original event.

Patterns I used: Prototype.

Issue 7 – Logger and Dashboard were hard-coded:

Inside `process()`, the processor manually called both:

1. `dashboard.updateMetrics(e)`

2. `logger.log(e)`

   Adding any new component requires changing the processor again, which breaks the Open-Closed Principle.

After Refactor:
Introduced the `EventObserver` interface and a list of observers.
Now the processor only calls `notifyObservers(e)` and doesn't care who receives the event.

Patterns used: Observer

3. Design Decision Questions:

Why each design pattern was chosen ?

Observer

I used Observer because I wanted the processor to notify the Dashboard and Logger automatically.
This way the processor doesn't depend on them directly.

Strategy

Each event type needs different behaviour.
Strategy lets me put every behaviour in a separate class instead of using if-else.

Template Method

The event transformation has a fixed order.
Template Method keeps this order clear and puts all steps in one place.

Decorator

I used Decorator to add encryption, compression, and metadata in a flexible way.
I can combine them easily without writing messy code.

Prototype

I used Prototype to work on a copy of the event.
This prevents changing the original event.

Proxy + Connection Pool

Proxy hides the database details from the processor.
The connection pool helps reuse DB connections instead of creating a new one each tim

What alternatives you considered?

1. Using if-else for event types → not flexible.
2. Doing all transformations inside the processor → too much code in one place.
3. Using a simple database class → no control over connections.

How the patterns interact ?

1. Processor uses Template Method to run the transformation.
2. Observers (Dashboard, Logger) get notified.
3. Decorator adds the extra transformations.
4. Prototype creates a safe event copy.

5. Strategy runs the correct behaviour for each event type.
6. Proxy + Pool handle database saving.

How your architecture improves scalability, flexibility, and maintainability?

Scalable

Connections are reused and the system handles more events easily.

Flexible

I can add new event types, observers, or transformations without changing the main code.

Easy to maintain

Each part has one job only, so the code is cleaner and easier to update.