Problems :

In Connection :

1. Create new connection No Reuse !

```
public Connection createConnection() {
   int n = ++counter;
   if (n > MAX) {
      System.out.println("Warning: too many connections created. Still creating (naive).");
      return null;
   }
   return new Connection("Conn-" + n);
}
```

2.Don’t stop at 10 , just warning messege !

```
if (n > MAX) {
   System.out.println("Warning: too many connections created. Still creating (naive).");
```

```
    return null;
}
```

3.Close connection dont do anything just print !

```
public void closeConnection(Connection c) {
    System.out.printf("[ConnectionManager] Closing connection %s (naive).%n", c.getId());
}
```

In Template:

1. Creates new Job from scratch — No cloning / No reuse

```
public Job createJobInstance() {
    String id = templateBody + " _ " + type + "-" + System.currentTimeMillis();
    return new Job(id, type, name, config);
}
```

2. misleading design:

```
private final String type;
```

private final String name;

private final String config;

private final String templateBody;

3. Open/Closed Principle (OCP):

String id = templateBody + " _ " + type + "-" +
System.currentTimeMillis();

4. Tight coupling with Job class

IN Job Executor:

1. Single Responsibility Violation
2. Long `if/else` on Job Type (Open/Closed Violation)
3. Tight Coupling to Connection Management

Why i use this design pattern :

Connection Pool: was used to limit database connections to a fixed pool, reduce creation overhead, and reuse existing connections instead of creating a new one per job.

Prototype: was chosen for job templates to avoid rebuilding large configurations from scratch and to quickly clone predefined templates for new jobs.

Strategy: was used to separate the execution logic of each job type (email, data, report) and remove type-based `if/else` checks from the executor.

Proxy: was chosen to add cross-cutting concerns (authorization, logging, timing, connection handling) around job execution without changing the core executor.

alternatives:

For job templates:

We could use a simple factory that creates a new object every time.

But this means we repeat the same configuration again and again, and it becomes slower because we always

create new objects.

For job execution:

We could keep everything inside one `executeJob` method and use `if/else` or `switch` to check the job type.

But this is not good, because the class becomes harder to change or extend if we add new job types.

For logging and permissions:

We could write logging, permission checks, and connection code directly inside `JobExecutor`.

But this mixes too many responsibilities in one class and makes the code harder to maintain.

How the patterns interact:

The **Prototype** layer creates `Job` instances from reusable templates.

The **Proxy** receives these jobs, validates the user, logs the execution, measures time, and manages connections using the **Connection Pool**.

The Proxy then delegates to the **JobExecutor**, which uses the **Strategy** pattern to pick the correct job strategy and run the business logic.

Together, the patterns form a clear pipeline: *Template* . *Proxy* . *Strategy* , all using shared pooled connections.

How the architecture improves scalability, flexibility, and maintainability:

Scalability: the Connection Pool allows many jobs to run without exhausting resources, and the Proxy centralizes resource management.

Flexibility: new job types can be added by creating new templates and strategies without modifying existing executor code.

Maintainability: responsibilities are clearly separated (templates, execution strategies, cross-cutting concerns, connection management), which reduces coupling and makes changes safer and easier to reason about.