Project: TaskMaster Processing System (TMPS) – Original Code Review
Design Decision Document

Hamza Zarour

1) Why each design pattern was chosen

Prototype Pattern (for job templates)
The system creates **heavy** job templates that simulate expensive initialization (simulateHeavyLoad). In the naive design, these templates were rebuilt from scratch every time we needed a new job, which wastes time and resources.
The Prototype pattern was chosen to:

- Build each heavy template only once.
- Store it as a prototype in a registry.
- Quickly clone it and create new Job instances without repeating the heavy work.

Strategy Pattern (for job execution)
Originally, JobExecutor contained long if/else chains based on string job types ("EMAIL", "DATA", "REPORT"). This design made the executor a “BIG class” and violated SRP and OCP.
The Strategy pattern was chosen to:

- Move the execution logic of each job type into its own class (EmailJobStrategy, DataProcessingStrategy, ReportGenerationStrategy).
- Keep JobExecutor focused on selecting and calling the correct strategy only.
- Make it easy to add new job types by adding new strategies instead of editing a big if/else.

Proxy Pattern (for controlled execution)
The original execution flow had no central point for permission checks, logging, timing, or connection handling. Adding these logics directly to JobExecutor would make it very complex.
The Proxy pattern was chosen to:

- Introduce JobExecutorProxy as a controlled front for all external executions.
- Perform permission checks based on User and job.getType().
- Log job start/end and measure execution time.
- Acquire and release connections from the pool in a safe way.
- Keep the original JobExecutor available for internal use if needed.

Connection Pool Pattern (for database connections)
The naive ConnectionManager always created new connections and could return null.
No real limit for connections. There was no real pooling or reuse.
The Connection Pool pattern was chosen to:

- Limit the number of active connections to a fixed maximum (10).
- Reuse existing connections using a blocking queue.
- Provide a simple logic way (acquire() / release()) that can be used by the Proxy without exposing low-level details.

2) What alternatives you considered

- **For Prototype:**
  I considered using a simple factory or builder to create jobs directly. However, this still required calling the heavy initialization each time or implementing manual caching logic. Prototype is a more natural and explicit way to express “build once, clone many times”.
- **For Strategy:**
  The alternative was to keep the existing if/else or to use a switch on the job type. Both options keep all logic in one class and break OCP, because every new job type would require changes inside JobExecutor. Strategy gives a cleaner separation and better extensibility.
- **For Proxy:**
  Another option was to put logging, permission checks, and timing directly inside JobExecutor. This would mix infrastructure concerns with business logic and make the class harder to maintain. Using a separate JobExecutorProxy keeps cross-cutting concerns isolated and reusable.
- **For Connection Pool:**
  I could have slightly modified ConnectionManager (like count connections with an integer), but that would still not provide safe reuse, blocking behavior, or a clear abstraction. A dedicated ConnectionPool based on a BlockingQueue is safer and closer to real-world designs.

3) How the patterns interact

The four patterns are designed to work together in a single execution flow:

1. **Prototype + Registry + TemplateManager**
   - At startup, TemplateManager builds each heavy template once and registers it in JobTemplateRegistry.
   - When the application needs a job, it calls TemplateManager.createJobFromTemplate(key).
   - The registry clones the stored prototype and creates a new Job instance.
2. **Job + User**
   - The created Job is linked to a User using setRequestedBy(user), which will later be used for permission checks in proxy.
3. **Proxy + Connection Pool + Executor + Strategy**
   - The client calls JobExecutorProxy.executeJob(job).
   - The Proxy:
     - Reads the User from the job and checks if the user has permission for job.getType().
     - Acquires a Connection from ConnectionPool.
     - Delegates to JobExecutor, passing the Job and Connection.
   - JobExecutor:
     - Asks JobStrategyFactory for the correct JobStrategy based on job.getType().
     - Calls strategy.execute(job, connection).
   - After execution, the Proxy releases the connection back to the pool and logs the execution time.

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

Scalability

- The Connection Pool ensures that only a limited number of connections are active and reused efficiently, which is important when many jobs are executed.
- Prototype reduces the cost of creating jobs from heavy templates, which improves performance under load.

Flexibility

- New job types can be added by:

  Creating a new template class (subclass of HeavyTemplate) and registering it in the registry.

  Creating a new JobStrategy and updating JobStrategyFactory.
- The Proxy can be extended to add more logics and validations without changing the job execution logic or templates.

Maintainability

- Responsibilities are clearly separated:
  - Templates and Prototypes handle job object creation.
  - Strategies handle how each job type is executed.
  - The Proxy handles permissions, logging, timing, and connections.
  - The Pool manages database resources.
- Classes are smaller and easier to test in isolation.
- The design follows SRP and OCP much better than the original version, which makes future changes less risky and easier to implement.