Design Decision Document

TaskMaster Processing System (TMPS) — Refactored Architecture
Advanced Software Development — Assignment 1

Introduction .1

The original TMPS implementation contained several architectural and object-oriented issues, including tight coupling, duplicated code, inefficient resource usage, and lack of extensibility.
The goal of the refactoring process was to transform the system into a clean, maintainable, :scalable solution using four mandatory design patterns

- Connection Pool
- Prototype
- Strategy
- Proxy

This document explains why each pattern was chosen, what alternatives were considered, .and how the patterns interact within the final architecture

---

Design Patterns Applied .2

(Prototype Pattern (Job Templates 2.1

Problem

Templates such as Email, Data Processing, and Report were recreated from scratch every time.
Template creation simulated heavy processing (3 seconds sleep), which caused unnecessary delays.
. There was no template reuse, no caching, and significant duplicated logic

Design Goal

- .Allow reuse of heavy templates
- .Avoid repeated expensive initialization
- .Enable fast cloning of pre-built templates

Chosen Solution: Prototype Pattern

:The following components were introduced

- JobPrototype interface
- HeavyTemplate abstract base
- Concrete prototypes: EmailJobTemplate, DataProcessingJobTemplate, ReportJobTemplate
- JobTemplateRegistry for storing reusable prototypes

The first time a template is requested, it is built and stored in the registry.
. Subsequent requests return lightweight clones of the stored prototype

Why Prototype Fits

- .Creation cost is high, cloning cost is low
- .Provides a centralized registry for managing reusable templates
- .Reduces template creation time and improves scalability

Alternatives Considered

Alternative	Reason for Rejection
Factory Method	.Still requires creating templates from scratch
Singleton per template	Does not support multiple independent job .instances

.Overly complex for this assignment JSON/Serialization caching

---

(Strategy Pattern (Job Execution Logic 2.2

Problem

:The original JobExecutor contained a long if/else chain to determine job behavior

```
... (if (type == EMAIL

... (else if (type == DATA

... (else if (type == REPORT
```

This caused tight coupling between job types and executor logic, violating the Open–Closed .Principle

Design Goal

- .Encapsulate job-specific algorithms
- .Allow new job types to be added without modifying JobExecutor
- .Improve readability, cohesion, and extensibility

Chosen Solution: Strategy Pattern

:Created

- JobStrategy interface
- EmailJobStrategy
- DataProcessingStrategy
- ReportGenerationStrategy
- JobStrategyFactory for type-to-strategy mapping

.JobExecutor simply selects the strategy and executes it, without any conditional logic

Why Strategy Fits

- .Clean separation between job execution algorithms
- .Eliminates branching logic
- .Easily extendable for new job types

Alternatives Considered

Pattern	Reason for Rejection
Command	.Adds unnecessary complexity
Template Method	.Does not fit because job algorithms differ significantly
Polymorphism inside Job	.Jobs should represent data, not behavior

---

(Connection Pool Pattern (Resource Management 2.3

Problem

:ConnectionManager in the original implementation

- .Created a new connection every time
- .Returned null when exceeding 10 connections instead of blocking
- .Lacked reuse and thread safety
- .Did not manage connection lifecycle correctly

Design Goal

- .Reuse a fixed number of connections

.Enforce a maximum of 10 active connections ●

.Block when no connection is available ●

.Ensure thread safety ●

Chosen Solution: Connection Pool

:Implemented a real connection pool using

A queue of available connections ●

A counter for created connections ●

()Blocking acquire() using wait ●

notify() when releasing connections ●

.ConnectionManager was refactored to serve as a simple facade over the pool

Why Connection Pool Fits

.Efficient and safe resource management ●

.Prevents uncontrolled creation of connections ●

.Supports concurrent execution ●

Alternatives Considered

Alternative	Reason for Rejection
Creating new connection per request	.Not scalable; wasteful
ExecutorService thread pool	.Not applicable to DB connections
Reactive resource management	.Too advanced and unnecessary

(Proxy Pattern (Controlled Job Execution 2.4

Problem

:Previously, jobs were executed without

- Permission checks
- Logging
- Execution timing
- Resource control
- Centralized monitoring

.JobExecutor was overloaded with responsibilities and violated SRP

Design Goal

- .Add a layer that handles authorization, logging, and timing
- .Manage connection lifecycle externally
- .Keep JobExecutor responsible only for executing strategy logic

Chosen Solution: Proxy Pattern

:Created JobExecutorProxy to

- Validate user permissions
- Log start/end execution
- Measure execution time
- Acquire and release connections
- Delegate actual work to JobExecutor

The real executor now only applies the strategy, while the proxy handles all cross-cutting .concerns

Why Proxy Fits

- .Adds functionality transparently without modifying original logic
- .Supports security and monitoring requirements
- .Matches the requirement that the original execute method must remain available

Alternatives Considered

Pattern	Reason for Rejection
Decorator	.Not ideal for permission/security concerns
Adapter	.Not relevant to the problem
AOP	.Too advanced; not in assignment scope

Interaction Between Patterns .3

:The system now follows a clean layered flow

.User submits a Job .1

JobExecutorProxy handles permission checks, logging, monitoring, and connection .2 .pooling

.JobExecutor receives the job and executes the correct JobStrategy .3

.TemplateManager supplies cloned templates using the Prototype pattern .4

.ConnectionPool provides database connections efficiently .5

.The architecture achieves strong separation of concerns and reduced coupling

---

Improvements Over Original .4 Architecture

Aspect	Before	After
Template creation	Slow, duplicated	Fast cloning via Prototype
Job execution	Hard-coded if/else	Clean Strategy pattern
Permissions	None	Proxy provides centralized validation
Logging & monitoring	None	Proxy handles all cross-cutting concerns
DB connections	Unlimited creation, unsafe	Connection pool with controlled reuse
Maintainability	Difficult	Highly modular and extensible
Adherence to SOLID	Poor	Strong compliance

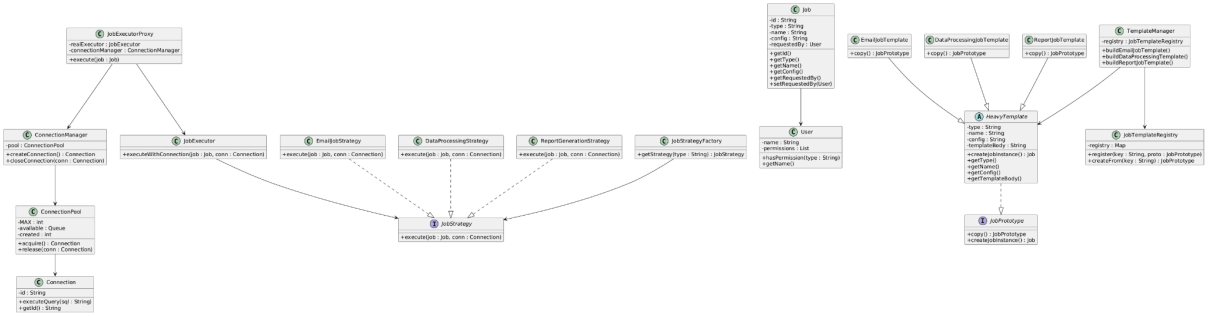
Conclusion .5

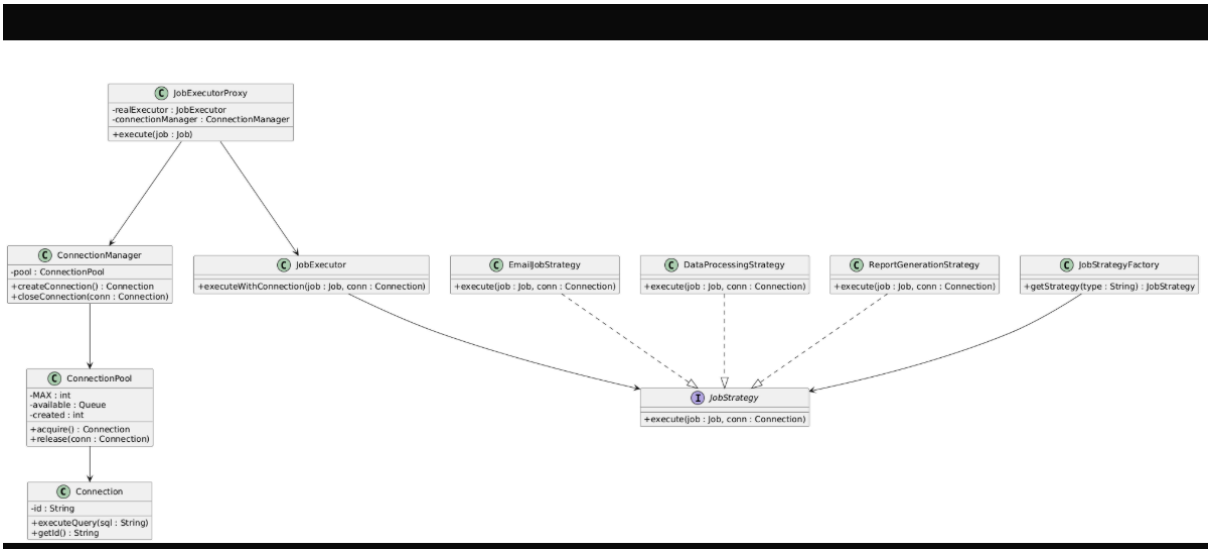
:The refactored TMPS architecture provides

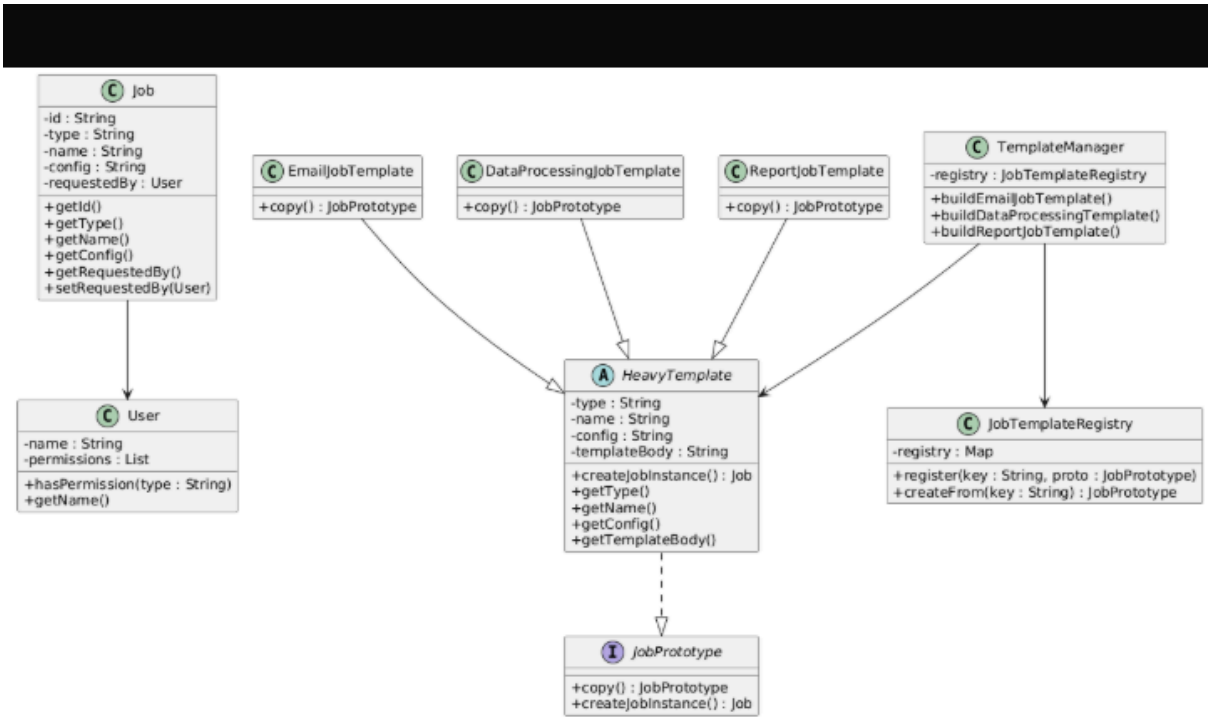
- Higher performance
- Improved modularity
- Reduced duplication

- Stronger adherence to object-oriented principles
- Centralized control for cross-cutting concerns
- Efficient resource utilization

The four design patterns work together cohesively to produce a scalable and maintainable processing system






Job
-id : String
-type : String
-name : String
-config : String
-requestedBy : User
+getId()
+getType()
+getName()
+getConfig()
+getRequestedBy()
+setRequestedBy(User)
EmailJobTemplate
+copy() : JobPrototype
DataProcessingJobTemplate
+copy() : JobPrototype
ReportJobTemplate
+copy() : JobPrototype
TemplateManager
-registry : JobTemplateRegistry
+buildEmailJobTemplate()
+buildDataProcessingTemplate()
+buildReportJobTemplate()
User
-name : String
-permissions : List
+hasPermission(type : String)
+getName()
HeavyTemplate
-type : String
-name : String
-config : String
-templateBody : String
+createJobInstance() : Job
+getType()
+getName()
+getConfig()
+getTemplateBody()
JobTemplateRegistry
-registry : Map
+register(key : String, proto : JobPrototype)
+createFrom(key : String) : JobPrototype
JobPrototype
+copy() : JobPrototype
+createJobInstance() : Job