CMPE202 - Individual Project

Part I (25 points)

Answer/outline the following:

- Describe what problem you're solving.
- What design pattern(s) will be used to solve this?
- Describe the consequences of using this/these pattern(s).
- Create a class diagram - showing your classes and the Chosen design pattern

---

1. Problem Description

The objective is to build a command-line Java application capable of parsing a heterogeneous log file containing APM logs, application logs, and request logs. The application classifies logs, aggregates key metrics, and outputs the results as structured JSON files. Each log type requires a different parsing and aggregation strategy. The solution must support extensibility for new log formats with minimal code changes.

2. Design Patterns Used:

2.1 Strategy Pattern

Usage - Implemented across different LogParser classes—ApmLogParser, ApplicationLogParser, and RequestLogParser—all implementing the common LogParser interface.
Purpose - Decouples parsing logic from orchestration logic, allowing each log type to define its own parsing rules.

2.2 Factory Pattern

Usage - The ParserFactory class returns a unified list of all log parsers.

Purpose - Centralizes parser registration, enabling scalable and maintainable addition of new log types.

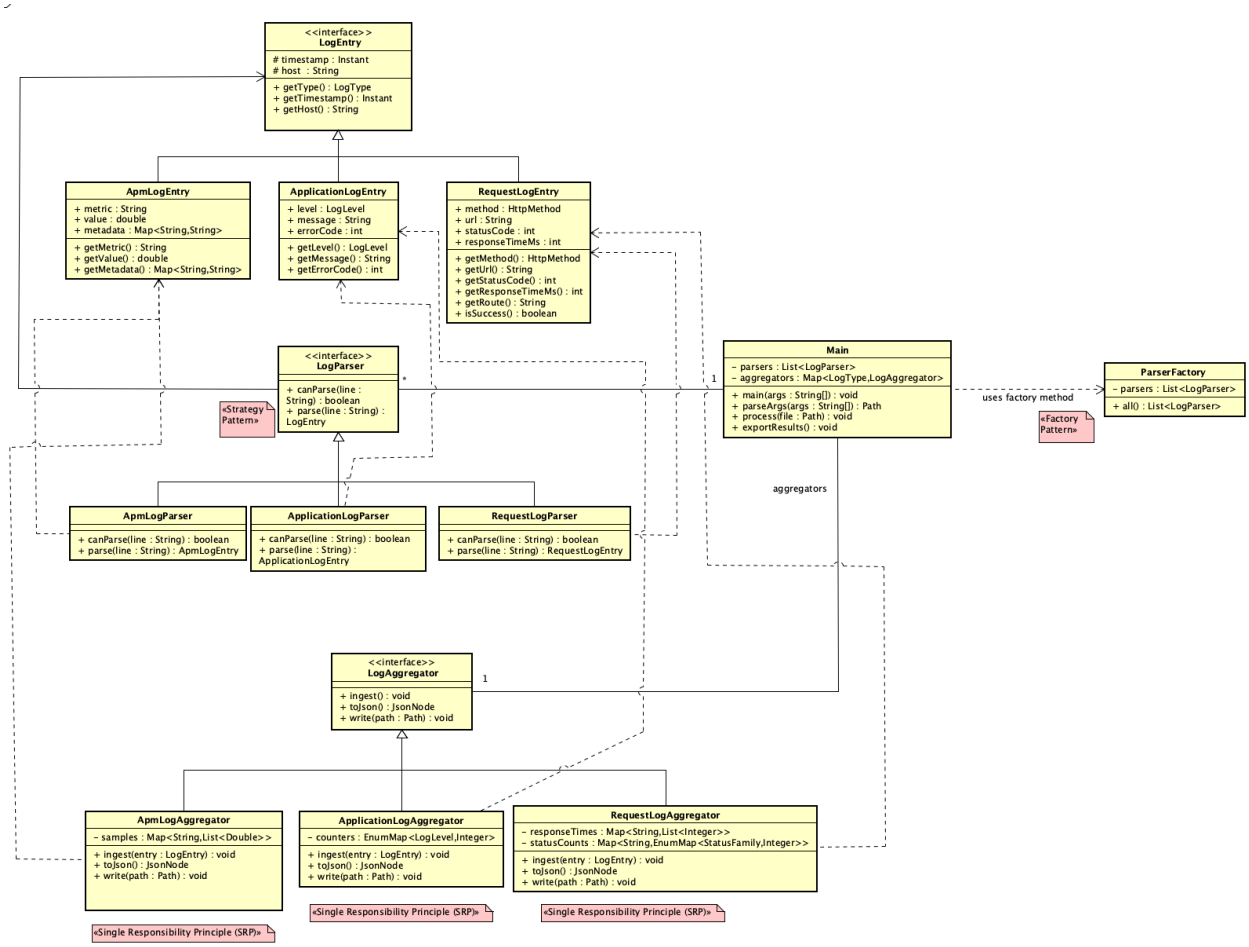
2.3 Single Responsibility Principle

Usage - Each component—parsers, log models, aggregators—has a focused responsibility.
Purpose - Ensures clear separation of concerns, better readability, and maintainability.

3. Consequences of Using These Patterns

Design Pattern	Benefits	Why It Helps	Trade-offs
Strategy	Easy to support new log types (e.g., audit logs) without changing existing logic	Each parser implements a shared interface, so they can be used interchangeably	More classes to maintain, even for simple logic
Factory	Centralizes parser registration in one place (ParserFactory)	The main app doesn't need to know parser details; it just loads them via the factory	Needs manual updates when adding new parsers
Single Responsibility Principle (SRP)	Modular, testable, and easier to maintain	Parsing, modeling, and aggregation are clearly separated	Can lead to a large number of small classes if not well organized

4. Create a class diagram - showing your classes and the Chosen design pattern



You can also find class diagram here: [Class Diagram]

Part II (75 points)

Implement an application (Java code and JUnit tests) for Part 1. Output should contain the details specified in the project description.

This part of the project involved developing a modular Java-based command-line application that parses a mixed log file and generates structured aggregations in JSON format for three log types: APM logs, Application logs, and Request logs.

Key Features Implemented

1. **Robust log parsing using Strategy Pattern**
   - Implemented LogParser interface with specialized classes: ApmLogParser, ApplicationLogParser, RequestLogParser.
   - Each parser detects and processes its respective log format.

2. **Centralized parser access using Factory Pattern**
   - ParserFactory provides a single point to access all registered parsers.

3. **Modular aggregation logic respecting SRP**
   Created one LogAggregator implementation per log type:

   - ApmLogAggregator computes min, median, average, max for metrics.

   - ApplicationLogAggregator counts logs by severity (INFO, ERROR, etc.).

   - RequestLogAggregator calculates response time percentiles and HTTP status code counts per endpoint.

4. **Main orchestration class**
   - The Main class handles file input, dispatches lines to all parsers, and routes LogEntry objects to all aggregators.Outputs written to apm.json, application.json, and request.json.

5. **Error-tolerant design**
   - Invalid or unsupported lines are silently ignored, ensuring resilience during log parsing.

Output Files

- apm.json – APM metric aggregations

- application.json – Log level counts

- request.json – API route response statistics

Testing

- **Unit tests** were implemented using **JUnit 5**, with one test class for each log aggregator:
    - ApmAggregatorTest validates correct computation of minimum, median, average, and maximum metric values.
    - ApplicationAggregatorTest verifies log severity counts for levels such as INFO, ERROR, and DEBUG.
    - RequestAggregatorTest ensures accurate calculation of response time percentiles and HTTP status code distributions per route.

- Each test feeds controlled input into the aggregator and compares the generated JSON snapshot against expected values using assertions.

Github Repo : [202 Individual Project]