Soumya Bharathi Vetukuri
016668964

CMPE-202 Individual Project– Part I

Log Parser Design Document

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 Statement

Modern applications produce diverse logs across multiple subsystems, including performance metrics, application status messages, and HTTP request logs. These logs are often unstructured, voluminous, and difficult to analyze manually. Without automated processing, extracting insights from logs is inefficient and error prone.

This project addresses this challenge by designing a command-line Java application that automates the parsing, classification, and aggregation of log entries from a unified .txt file. Each log line is categorized into one of three supported types:

- APM Logs: Contain metrics like CPU, memory, disk usage, and more.
- Application Logs: Represent log levels like ERROR, INFO, WARNING, DEBUG.
- Request Logs: Detail API requests, including method, endpoint, response time, and status code.

The system computes and outputs the following structured aggregations:

- APM Logs: Min, median, average, and max per metric.
- Application Logs: Count of log entries grouped by severity level.
- Request Logs: Percentile statistics and HTTP status counts per route.

Each category's output is saved to its own JSON file (apm.json, application.json, request.json), enabling further visualization, analytics, or alerting pipelines. The system is extensible for future support of additional log types and formats.

Soumya Bharathi Vetukuri
016668964

2. Design Patterns Employed

To build a maintainable and extensible solution, three core design patterns were implemented:

1. Strategy Pattern

- Purpose: Encapsulate the parsing logic for each log type.
- Implementation: The LogParser interface is implemented by concrete parser classes: APMLogParser, ApplicationLogParser, and RequestLogParser. This allows the parsing logic for each log type to vary independently without affecting others.
- Benefit: Enables polymorphic behavior while maintaining single-responsibility logic for each parser.

2. Factory Pattern

- Purpose: Select and instantiate the correct parser class based on the structure of the log line.
- Implementation: The LogParserFactory uses conditional checks to return the appropriate LogParser implementation based on line characteristics.
- Benefit: Simplifies the main application logic and supports Open/Closed Principle for adding new log types.

3. Utility Pattern (Singleton-like)

- Purpose: Centralize stateless functionality like JSON serialization and statistical computation.
- Implementation: JsonWriterUtil and StatUtils are implemented with static methods only.
- Benefit: Promotes reuse, avoids global state, and simplifies testing and access.

3. Consequences of Using These Patterns

1. Strategy Pattern

- **Advantages**:
  - Open/Closed Principle: New log types (example: SecurityLogParser) can be added by implementing LogParser without altering existing logic.
  - Modularity: Each parser class encapsulates its own logic, making debugging and enhancements easier.
  - Testability: Individual parsers can be unit tested independently.
  - Reusability: Parser strategies can be reused in other parsing pipelines if needed.

- **Trade-offs**:
    - Increased Class Count: For every new log type, a new parser class must be created.
    - Complexity Overhead: May seem heavy-handed for small or one-off projects.
    - Developer Familiarity: Requires understanding of interface-based programming and polymorphism.

2. Factory Pattern

- **Advantages**:
    - Centralized Control: Creation logic is centralized, improving readability and maintainability.
    - Extensibility: Adding new log types only requires updating the factory's parser selector logic.
    - Abstraction: Shields the main application (LogParserMain) from needing to know about parser implementation details.
- **Trade-offs**:
    - Conditional Logic: As the number of log types grows, factory logic can become bloated.
    - Indirection: Adds a layer of abstraction that may slow understanding for new developers.

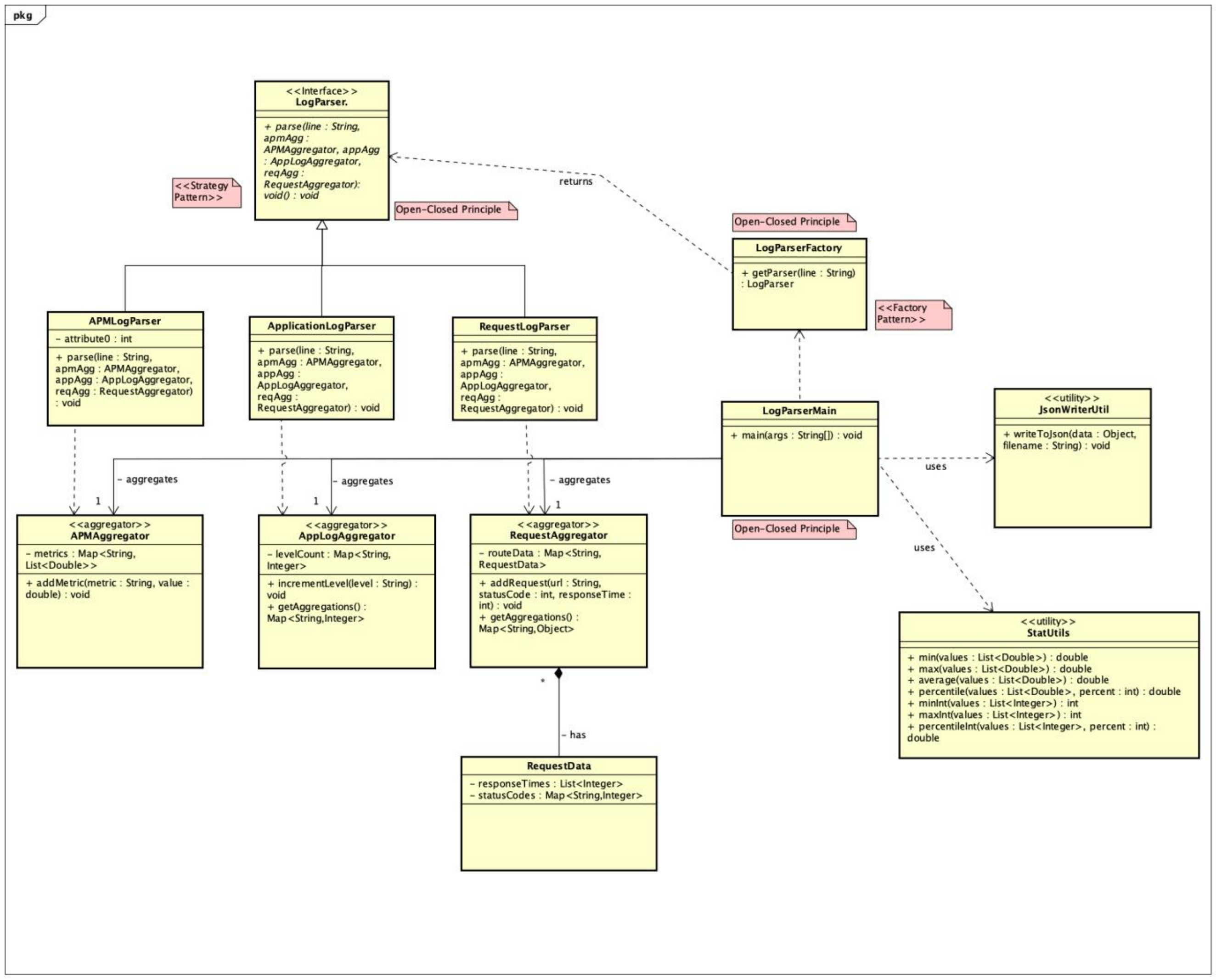
3. Utility Pattern (StatUtils, JsonWriterUtil)

- **Advantages**:
    - Shared Functionality: Eliminates redundant code by centralizing common tasks.
    - Statelessness: Static methods mean no side effects or retained state — safe for concurrent use.
    - Simplicity: Easily accessible methods simplify statistical and serialization logic.
- **Trade-offs**:
    - Limited Extensibility: If logic becomes complex or stateful in the future, transitioning from static methods may be non-trivial.
    - Testing Limitations: Testing static methods may require additional frameworks, although current logic is simple enough not to need mocking.

Soumya Bharathi Vetukuri
016668964

4. UML Class Diagram

The following diagram shows the key classes, relationships, and pattern annotations in the system:


pkg
<<Interface>>
LogParser.
+ parse(line : String, apmAgg : APMAggregator, appAgg : AppLogAggregator, reqAgg : RequestAggregator): void() : void
<<Strategy Pattern>>
Open-Closed Principle
returns
Open-Closed Principle
LogParserFactory
+ getParser(line : String) : LogParser
<<Factory Pattern>>
APMLogParser
- attribute() : int
+ parse(line : String, apmAgg : APMAggregator, appAgg : AppLogAggregator, reqAgg : RequestAggregator) : void
ApplicationLogParser
+ parse(line : String, apmAgg : APMAggregator, appAgg : AppLogAggregator, reqAgg : RequestAggregator) : void
RequestLogParser
+ parse(line : String, apmAgg : APMAggregator, appAgg : AppLogAggregator, reqAgg : RequestAggregator) : void
LogParserMain
+ main(args : String[]) : void
Open-Closed Principle
<<utility>>
JsonWriterUtil
+ writeToJson(data : Object, filename : String) : void
uses
uses
- aggregates
1
- aggregates
1
- aggregates
1
<<aggregator>>
APMAggregator
- metrics : Map<String, List<Double>>
+ addMetric(metric : String, value : double) : void
<<aggregator>>
AppLogAggregator
- levelCount : Map<String, Integer>
+ incrementLevel(level : String) : void
+ getAggregations() : Map<String,Integer>
<<aggregator>>
RequestAggregator
- routeData : Map<String, RequestData>
+ addRequest(url : String, statusCode : int, responseTime : int) : void
+ getAggregations() : Map<String,Object>
*
- has
RequestData
- responseTimes : List<Integer>
- statusCodes : Map<String,Integer>
<<utility>>
StatUtils
+ min(values : List<Double>) : double
+ max(values : List<Double>) : double
+ average(values : List<Double>) : double
+ percentile(values : List<Double>, percent : int) : double
+ minInt(values : List<Integer>) : int
+ maxInt(values : List<Integer>) : int
+ percentileInt(values : List<Integer>, percent : int) : double

- Interface: LogParser
- Parsers: APMLogParser, ApplicationLogParser, RequestLogParser
- Aggregators: APMAggregator, AppLogAggregator, RequestAggregator
- Utilities: StatUtils, JsonWriterUtil
- Factory: LogParserFactory
- Entry Point: LogParserMain
- Composition: RequestAggregator owns RequestData
- Multiplicity, Labels: aggregates, uses, returns, has