Software Systems Engineering CMPE 202 Individual Project

Name: Deepak Kumar Yuvaneesh Pradeepkumar

SJSU ID: 018194072

1. Problem Description

The project solves the problem of processing a log file containing multiple types of log entries and generating aggregated JSON reports for each log type.

Functionalities:

1. **Parsing Logs:**
    - Parse log files containing multiple types of logs (APM, Application, Request).
    - Extract key information based on log structure.
2. **Classifying Logs:**
    - Classify logs into categories (APM, Application, Request).
3. **Aggregating Logs:**
    - APM Logs: Calculate min, max, median, and average for metrics like CPU and memory usage.
    - Application Logs: Count entries by severity level (e.g., INFO, DEBUG, ERROR).
    - Request Logs: Compute response time statistics and categorize HTTP status codes by API route.
4. **Generating Output:**
    - Write JSON reports into separate files for each log type (apm.json, application.json and request.json).

Future Extensibility:

1. **New Log Types:** The design should allow adding new types of logs (e.g., Security Logs).
2. **Additional File Formats:** The application should be extensible to support other input/output file formats like CSV.

2. Design Patterns Used

a. Factory Method Pattern

- **What it does:**

Decouples the creation of objects (parsers, aggregators, file processors) from their usage.

- **How it's used:**

1. **LogParserFactory:**
    - Creates the appropriate parser for each log type (e.g., APMLogParser, RequestLogParser) based on the log structure.
2. **LogAggregatorFactory:**
    - Creates the appropriate aggregator (e.g., APMAggregator, RequestAggregator) for each log type.
3. **FileProcessorFactory:**
    - Selects the appropriate file processor implementation (e.g., TextFileProcessor, JsonFileProcessor) based on the file extension.
    - Ensures future extensibility by allowing additional file formats (e.g., CSV) without modifying the FileHandler.

b. Strategy Pattern

- **What it does:** Encapsulates parsing and aggregation logic into separate strategies for each log type.
- **How it's used:**
    - LogParser is an abstract class with concrete implementations for each log type (e.g., APMLogParser, RequestLogParser).
    - LogAggregator is an abstract class with concrete implementations for aggregating each log type (e.g., APMAggregator, RequestAggregator).

c. Singleton Pattern

- **What it does:** Ensures a single instance of FileHandler is responsible for file operations, such as reading the input log file and writing JSON output files.
- **How it's used:** The FileHandler class is implemented as a singleton, preventing multiple instances from being created.

3. Consequences of Using These Patterns

a. Factory Method Pattern

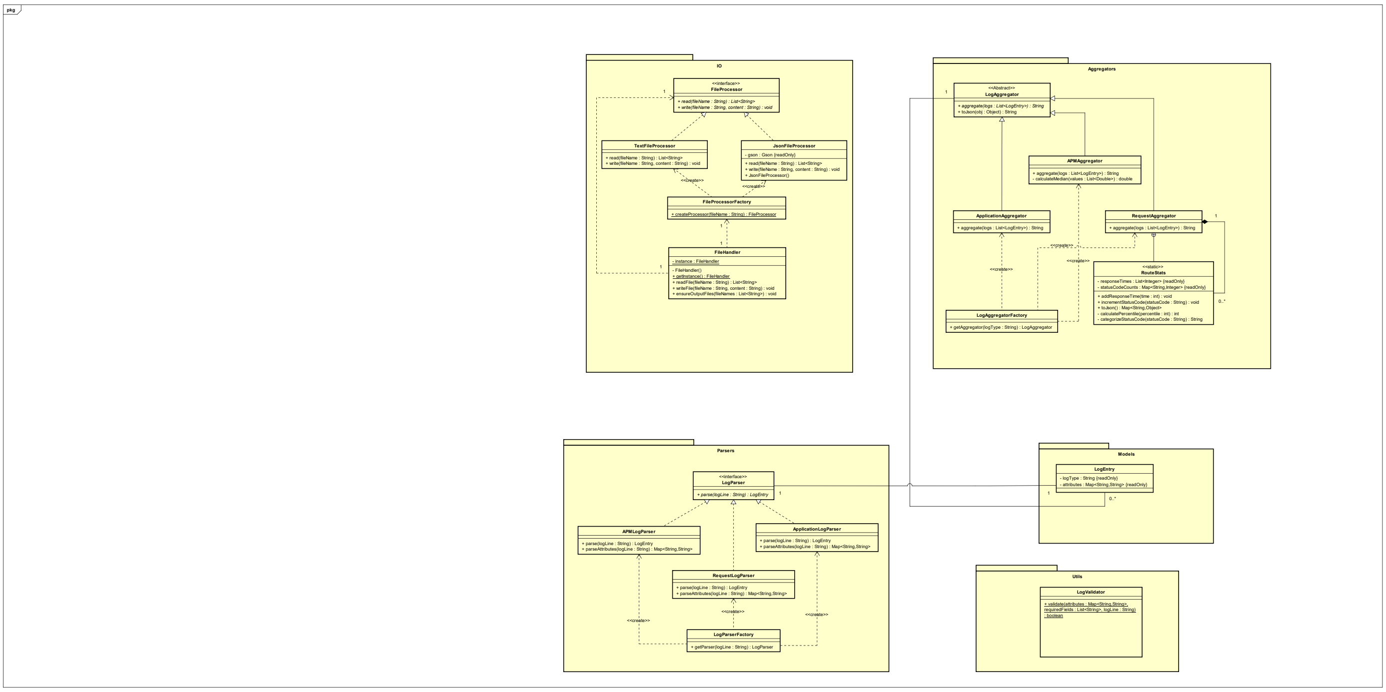
Advantages	**Disadvantages**
Encapsulates object creation logic, keeping the main application clean.	Factories require updates when new log types or file formats are introduced, violating OCP.
Adding a new parser, aggregator, or file processor is straightforward.	Long if-else chains may become harder to maintain as log types grow.
Decouples parser/aggregator instantiation and file reader selection from the main application.	Tight coupling between factories and the types they create.

b. Strategy Pattern

Advantages	**Disadvantages**
Encapsulates parsing and aggregation logic for each log type in separate classes.	Requires creating two new classes (parser and aggregator) for every log type.
Highly modular—individual log types can be modified independently.	Runtime selection of strategy adds slight performance overhead.
New log types can be added without affecting existing ones, adhering to OCP.	Increased class count for each new log type can make the structure verbose.

c. Singleton Pattern

Advantages	**Disadvantages**
Ensures consistent file handling by using a single instance of FileHandler.	Introduces global state, which can complicate testing and debugging.
Reduces resource usage (e.g., multiple threads accessing the same file).	Limits flexibility—cannot use dependency injection easily.


pkg
IO
<<interface>>
FileProcessor
+ read(fileName : String) : List<String>
+ write(fileName : String, content : String) : void
TextFileProcessor
+ read(fileName : String) : List<String>
+ write(fileName : String, content : String) : void
JsonFileProcessor
- gson : Gson {readOnly}
+ read(fileName : String) : List<String>
+ write(fileName : String, content : String) : void
+ JsonFileProcessor()
<<create>>
<<create>>
FileProcessorFactory
+ createProcessor(fileName : String) : FileProcessor
FileHandler
- instance : FileHandler
- FileHandler()
+ getInstance() : FileHandler
+ readFile(fileName : String) : List<String>
+ writeFile(fileName : String, content : String) : void
+ ensureOutputFiles(fileNames : List<String>) : void
1
1
1
1
Aggregators
<<Abstract>>
LogAggregator
+ aggregate(logs : List<LogEntry>) : String
+ toJson(obj : Object) : String
APMAggregator
+ aggregate(logs : List<LogEntry>) : String
- calculateMedian(values : List<Double>) : double
ApplicationAggregator
+ aggregate(logs : List<LogEntry>) : String
RequestAggregator
+ aggregate(logs : List<LogEntry>) : String
<<static>>
RouteStats
- responseTimes : List<Integer> {readOnly}
- statusCodeCounts : Map<String,Integer> {readOnly}
+ addResponseTime(time : int) : void
+ incrementStatusCode(statusCode : String) : void
+ toJson() : Map<String,Object>
- calculatePercentile(percentile : int) : int
- categorizeStatusCode(statusCode : String) : String
LogAggregatorFactory
+ getAggregator(logType : String) : LogAggregator
<<create>>
<<create>>
<<create>>
1
1
0..*
Parsers
<<interface>>
LogParser
+ parse(logLine : String) : LogEntry
APMLogParser
+ parse(logLine : String) : LogEntry
+ parseAttributes(logLine : String) : Map<String,String>
ApplicationLogParser
+ parse(logLine : String) : LogEntry
+ parseAttributes(logLine : String) : Map<String,String>
RequestLogParser
+ parse(logLine : String) : LogEntry
+ parseAttributes(logLine : String) : Map<String,String>
LogParserFactory
+ getParser(logLine : String) : LogParser
<<create>>
<<create>>
<<create>>
1
Models
LogEntry
- logType : String {readOnly}
- attributes : Map<String,String> {readOnly}
1
0..*
Utils
LogValidator
+ validate(attributes : Map<String,String>, requiredFields : List<String>, logLine : String) : boolean