CMPE 202 Individual Project Part 1

Problem Description:

The core problem is to process a stream of diverse log entries from a text file, each representing different aspects of an application's operation and performance. The application needs to:

- **Parse:** Interpret each log line to identify its type and extract relevant data.
- **Categorize:** Classify logs into predefined types (APM, Application, Request).
- **Aggregate:** Perform specific calculations and data summarizations based on the category of the log. For example, calculating min/median/average/max for APM metrics, counting log levels for Application logs, and calculating response time percentiles and status code counts for Request logs.
- **Output:** Generate structured JSON files for each log category, containing the aggregated data.
- **Extensibility:** The design should be flexible enough to easily accommodate new log types and potentially new input file formats in the future without major overhauls to the existing system.
- **Error Handling:** Gracefully ignore corrupted or incompatible log lines.

The application will be a command-line tool, taking the input log file name as an argument and producing separate JSON output files for each log category.

Design Pattern:

Based on the problem description the primary design pattern I used is the **Chain of Responsibility** pattern.

Additionally, to handle the creation and management of different log aggregations, each handler is associated with a specific aggregator (ApmAggregator, ApplicationAggregator, RequestAggregator). Each handler knows how to process a specific type of log and uses its dedicated aggregator to store and compute the results.

Consequences of Using these Patterns : Why Chain of Responsibility is used ?

- ○ **Handling Diverse Log Types:** The input log file contains different types of entries (APM, Application, Request). Each type requires distinct parsing and processing logic.Chain of Responsibility was used as we don’t know which type a line belongs to before processing it .This pattern creates a chain of processing objects (handlers). Each handler contains the logic to process a specific type of request (in this case, a log line). When a request arrives, it's passed along the chain until one of the handlers processes it.

  - **Decoupling Senders and Receivers:** The main loop reading the file (the sender of the log line) doesn't need to know which specific handler will process the line. It simply passes the log line to the beginning of the chain. This makes the main loop simpler and more focused on I/O.

  - **Extensibility for New Log Types:** This is a key requirement. If you need to support a new log type in the future (e.g., "Security Logs"), you can create a new ‘SecurityLogHandler’ and easily insert it into the chain without modifying the existing handlers or the main processing loop significantly. This promotes the Open/Closed Principle (open for extension, closed for modification).

  - **Orderly Processing:** You can define the order in which handlers attempt to process a log line. So if we were also given that these types of logs are in majority , they can be put at the start of the chain.

  - **Single Responsibility Principle:** Each handler focuses solely on one type of log, making the code for each handler cleaner, easier to understand, and test.

- **Disadvantages**:

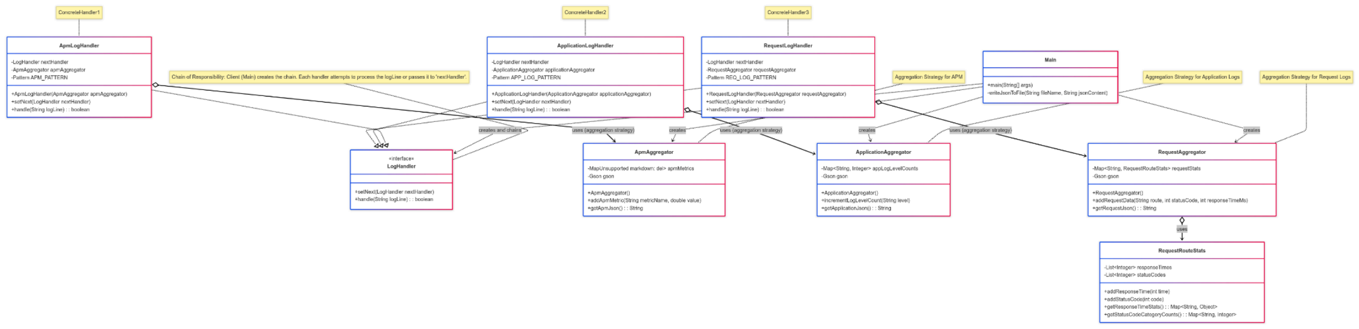
  - **No Guarantee of Handling:** A request (log line) might not be handled if it reaches the end of the chain and no handler has processed it. The project description addresses this by stating "Our application will simply ignore these" (corrupted/incompatible lines), and the Main.java code includes a warning for unhandled lines.

- **Potential for Performance Issues:** If the chain is long or handlers perform complex processing before deciding to pass the request on, it could lead to performance overhead as the log line traverses multiple handlers. For this specific problem, with a small number of well-defined log types, this is less likely to be a major issue.

- **Debugging Complexity:** Tracing a request through a long chain can sometimes be more complex to debug than direct method calls.

- **Order Dependency:** The order of handlers in the chain can be crucial. For instance, if one handler's matching criteria is a subset of another's, the more specific handler should typically come first.

<u>Class Diagram:</u>



ConcreteHandler1
ApmLogHandler
-LogHandler nextHandler
-ApmAggregator apmAggregator
-Pattern APM_PATTERN
+ApmLogHandler(ApmAggregator apmAggregator)
+setNext(LogHandler nextHandler)
+handle(String logLine) : : boolean
Chain of Responsibility: Client (Main) creates the chain. Each handler attempts to process the logLine or passes it to 'nextHandler'.
ConcreteHandler2
ApplicationLogHandler
-LogHandler nextHandler
-ApplicationAggregator applicationAggregator
-Pattern APP_LOG_PATTERN
+ApplicationLogHandler(ApplicationAggregator applicationAggregator)
+setNext(LogHandler nextHandler)
+handle(String logLine) : : boolean
ConcreteHandler3
RequestLogHandler
-LogHandler nextHandler
-RequestAggregator requestAggregator
-Pattern REQ_LOG_PATTERN
+RequestLogHandler(RequestAggregator requestAggregator)
+setNext(LogHandler nextHandler)
+handle(String logLine) : : boolean
Aggregation Strategy for APM
Main
+main(String[] args)
-writeJsonToFile(String fileName, String jsonContent)
Aggregation Strategy for Application Logs
Aggregation Strategy for Request Logs
creates and chains
uses (aggregation strategy)
creates
uses (aggregation strategy)
creates
uses (aggregation strategy)
creates
«interface»
LogHandler
+setNext(LogHandler nextHandler)
+handle(String logLine) : : boolean
ApmAggregator
-MapUnsupported markdown: del> apmMetrics
-Gson gson
+ApmAggregator()
+addApmMetric(String metricName, double value)
+getApmJson() : : String
ApplicationAggregator
-Map<String, Integer> appLogLevelCounts
-Gson gson
+ApplicationAggregator()
+incrementLogLevelCount(String level)
+getApplicationJson() : : String
RequestAggregator
-Map<String, RequestRouteStats> requestStats
-Gson gson
+RequestAggregator()
+addRequestData(String route, int statusCode, int responseTimeMs)
+getRequestJson() : : String
uses
RequestRouteStats
-List<Integer> responseTimes
-List<Integer> statusCodes
+addResponseTime(int time)
+addStatusCode(int code)
+getResponseTimeStats() : : Map<String, Object>
+getStatusCodeCategoryCounts() : : Map<String, Integer>