CMPE 202 Individual Project Part 1

Name: Bhavika Sodagum
SJSU ID: 017506567
Course: CMPE 202 - Software Systems Engineering

Problem Statement:

We need to develop a command-line application that will perform the following tasks:

1. Read log entries from a text input file
2. Parses and classifies the log data and categorises them as APM logs or Application logs or Request Logs
3. Performs Aggregations:
   a) APM Logs: Provides metrics such as min, median, average, max for every performance metric type like CPU, memory.
   b) Application Logs: Counts the number of log entries for each severity level, such as ERROR, INFO.
   c) Request Logs: Aggregates response times (min, max, percentiles) and counts HTTP status codes by type, such as 2XX, 4XX.
4. Writes the aggregated results(outputs) into separate JSON files (apm.json, application.json, request.json).
5. It should support extensibility, that is, it should be designed to easily accommodate additional log types and formats in the future.

Additionally, we need to ensure the application is tested with appropriate unit tests to verify all functionalities.

Design Patterns used to solve this problem:

1) Chain of Responsibility:
   - The Chain of Responsibility design pattern is used here to parse and process the different types of logs. A series of handlers namely - APMLogHandler, ApplicationLogHandler, and RequestLogHandler have been implemented where each handler is capable of handling a specific log type. If a handler is unable to process a log line, then it will pass the line to the next handler in the chain.
   - The handle() method is defined by the abstract base class Loghandler(), while the concrete classes (APMLogHandler, ApplicationLogHandler, and RequestLogHandler) provide specific implementations to process the log data.

2) Strategy Pattern:
   - The strategy pattern is used for aggregating metrics from parsed log entries. Each log type has an appropriate aggregator, such as APMLogAggregator, ApplicationLogAggregator, RequestLogAggregator, which provides the appropriate method to calculate statistics like averages, medians, and percentiles.

- This pattern enables flexible strategies for aggregation in each log type in order to make it easy to modify or extend the logic for aggregations independently of the parsing logic.

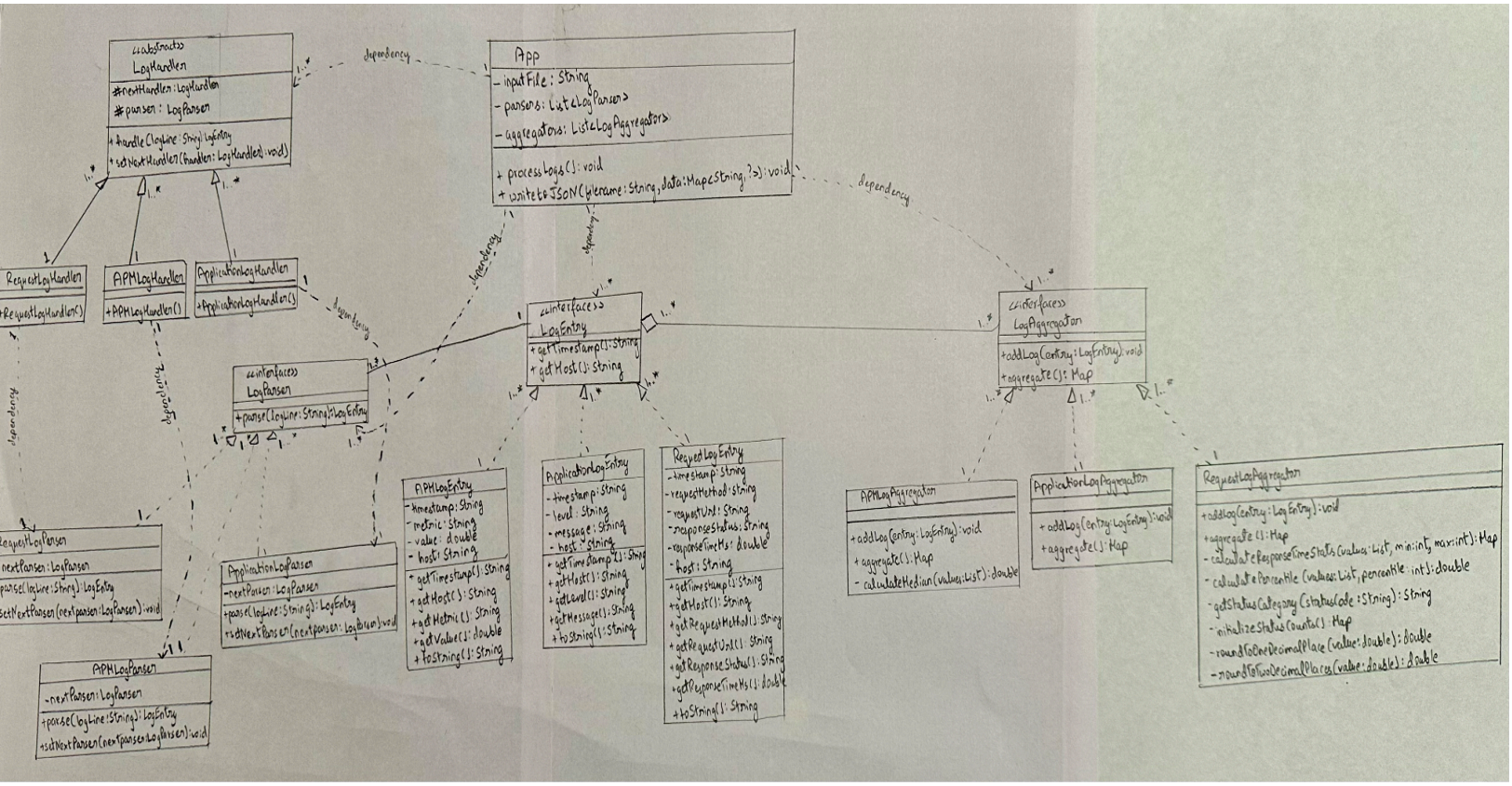
Consequences of using these patterns:

1. Chain Of Responsibility:
   Pros:
   - Simplifies code as this decouples the sender (log file reader) from the specific handlers.
   - This makes the system easily extensible by adding new handlers without changing the existing ones.

   Cons:
   - Can increase complexity if the chain becomes long enough.
   - This may lead to less efficient processing if the log type is frequently handled by the last handler in the chain.
2. Strategy Pattern:
   Pros:
   - Provides flexibility in defining various aggregation algorithms for various log types.
   - Easily add or change the aggregation logic for a particular log type without affecting the other logs.

   Cons:
   - Increased number of classes that may add to system complexity.
   - Requires careful coordination between the log parsers and aggregators to ensure correctness.

Class Diagram:



Explanation:

The components of the diagram can be broken down as follows:

1. Log Parsers (Chain of Responsibility Pattern):

- The LogParser interface serves as a contract that defines the method signature to parse logs (parse(logLine: String)).
- Concrete classes (APMLogParser, ApplicationLogParser, RequestLogParser) implement the LogParser interface, each responsible for parsing a specific type of log entry (APM logs, Application logs, Request logs).

2. The LogHandler is an abstract class that represents a Chain of Responsibility.

- Each LogHandler (APMLogHandler, ApplicationLogHandler, RequestLogHandler) will process log lines depending on the log type. Each handler can pass the log line to the next handler in the chain if it cannot handle the specific type.

- One main point of the Chain of Responsibility pattern here is that one might be allowed to handle the responsibility for parsing by different handlers without stating which handler will take care of a given log type; this makes the system flexible and easy to extend by addition of new types of log handlers.

Chain of Responsibility Workflow:

- LogHandler contains a reference (nextHandler) to another LogHandler, creating a chain.
- Each handler attempts to parse the log entry. If the current handler cannot handle it, the responsibility is passed to the nextHandler.
- This chain of log parsers allows the system to handle multiple types of log entries in a dynamic and decoupled manner, thus providing more scalability.

3. Log Entries:

- The LogEntry interface defines common behavior across all log entries (getTimestamp(), getHost()).
- Specific log entry classes (APMLogEntry, ApplicationLogEntry, RequestLogEntry) implement the LogEntry interface.
- This allows for a consistent way of handling different log entries throughout the application, regardless of their specific type, supporting polymorphism.

4. Log Aggregators (Strategy Pattern):

- It uses the Strategy Pattern to encapsulate aggregation logic for various log types.
- The LogAggregator interface declares methods to aggregate entries and calculate statistics (aggregate(entries: List<LogEntry>), calculateStatistics()).
- Concrete classes (APMLogAggregator, ApplicationLogAggregator, RequestLogAggregator) implement the LogAggregator interface by providing different specific aggregation strategies for different log types.
- This pattern enables the different log aggregation strategies to be interchanged. In the future, when new types of logs are added, new aggregator classes can be implemented without affecting the existing system, improving modularity and extensibility.

Strategy Pattern Workflow:

- The App class is responsible for log processing and JSON output.
- It uses different LogAggregator instances, depending on the type of log it has to process.
- This decouples the logic of aggregating logs from the rest of the application and allows new types of aggregations to be introduced in the future.

5. App Class:

- App class is the controller of the application.
- It holds references to the parsers - LogParser and the aggregator LogAggregator - that process and aggregate logs.
- The App class orchestrates the reading of the log file, parsing of the log entries, its aggregation, and finally writing the aggregation results to respective JSON files.

- It provides methods like processLogs() to parse the logs using the handlers and writeToJSON() to store the aggregated results.