Log Parser Application - Design Pattern Analysis

1. Describe what problem you're solving

The log parser application solves the problem of parsing a log file containing different types of log entries (APM logs, Application logs, and Request logs), categorizing these logs, and generating aggregated statistics for each log type. The application needs to:

- Parse and identify different log types within a single file
- Process each log type using specific algorithms
- Aggregate data for each log type according to different requirements:
  - For APM logs: Calculate min, median, average, and max for each metric
  - For Application logs: Count logs by severity level
  - For Request logs: Calculate response time statistics and count status codes by category
- Output the results to separate JSON files
- Support future extension for additional log types and file formats

2. What design pattern(s) will be used to solve this?

Three design patterns will be used to solve this problem:

1. **Strategy Pattern**: For implementing different log parsing algorithms for each log type
2. **Factory Pattern**: For creating appropriate parser instances based on log type
3. **Chain of Responsibility Pattern**: For determining which parser should handle each log entry

3. Describe the consequences of using this/these pattern(s)

Strategy Pattern Consequences:

Positive:

- Allows different parsing algorithms for each log type
- New log types can be added by implementing new parser strategies
- Parsing logic is separated from log processing logic
- Individual strategies can be tested in isolation

Negative:

- Increases the number of classes in the system
- May add complexity for simple parsing tasks

Factory Pattern Consequences:

Positive:

- Centralizes parser creation logic
- Decouples clients from concrete parser implementations
- New parser types can be added without changing client code

Negative:

- Adds another layer of abstraction
- Requires additional factory classes

Chain of Responsibility Pattern Consequences:

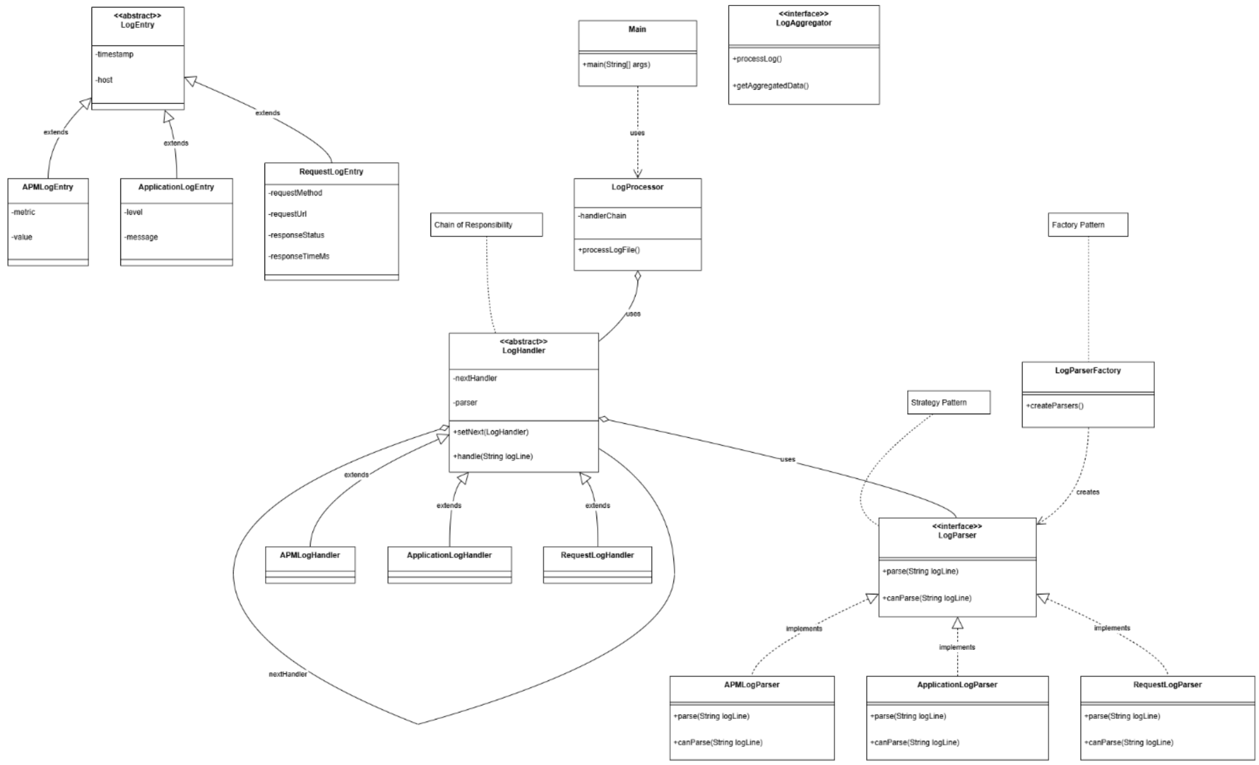
Positive:

- Decouples sender (log processor) from receivers (log handlers)
- Allows flexible configuration of the processing order
- Each handler focuses on processing one type of log
- Invalid logs can be gracefully ignored

Negative:

- No guarantee that a log entry will be processed if no handler matches
- Performance impact as logs may need to pass through multiple handlers
- Can make debugging more challenging

Class Diagram - Showing Classes and Chosen Design Patterns



Class Diagram Description

- **Strategy Pattern**: Implemented through the LogParser interface with concrete strategies (APMLogParser, ApplicationLogParser, RequestLogParser) providing different parsing algorithms for each log type.

- **Factory Pattern**: Implemented via LogParserFactory which centralizes parser creation, returning concrete implementations through the abstract interface.

- **Chain of Responsibility Pattern**: Implemented through the LogHandler abstract class and its subclasses, forming a chain where each handler attempts to process logs or passes them to the next handler.

- The diagram shows how these patterns interact with the LogEntry hierarchy and supporting classes to create a flexible, extensible system for parsing different log types.