Log Parser Application Design Document

Problem Description

The problem is that we must develop a command-line application as well as it must process log files since they contain different types of log entries. The application needs to:

1. Parse through a log file that has a .txt extension and also contains multiple types of log entries such as APM logs, Application logs, and Request logs.
2. Make a classification for each log entry so that it is inside its category.
3. For each log type, specific aggregations can be performed. Do this task for each type of log.
4. Compute minimum, median, average, and maximum APM logs values for each performance metric.
5. By severity level, application logs must be counted. These levels include ERROR, WARNING, INFO, and then DEBUG.
6. Calculate response time statistics like min, max, and percentiles for Request logs; count requests when you categorize by status code per API route.
7. For each log type, write aggregated data into separate JSON output files.
8. Extensibility should include support for more file formats plus log types.
9. Be sure that you ignore log entries that are corrupted or incompatible by the handling of them gracefully.

Design Patterns Used

To solve this problem efficiently and create an extensible design, I've used the following design patterns:

1. Strategy Pattern

The Strategy pattern encapsulates different log parsing strategies for use. The LogEntryParser interface implements this as well as does its concrete implementations. These do include APMLogParser, ApplicationLogParser, as well as RequestLogParser.

Purpose: Separate parsing logic for log types along with allowing new log types with code changes that are minimal.

2. Factory Method Pattern

The Factory Method pattern makes the suitable parser based on the log entry type. The LogParserFactory class does implement this one.

Reason: Object centralization lets the system specify the right parser when running.

3. Observer Pattern

The Observer pattern suits the aggregation process's use. The LogAggregator interface as well as its concrete implementations such as APMAggregator, ApplicationAggregator, and RequestAggregator implement this.

Purpose: To collect as well as to process log entries independently for each log type, which allows for new aggregators to be added without any change to the core parsing logic.

4. Template Method Pattern

The Template Method pattern standardizes the overall workflow for log processing. LogProcessor class is how the standardization occurs.

Purpose: We define the skeleton for the parsing process, subclasses can override only specific steps, and this reduces code duplication.

Consequences of Using These Patterns

Advantages

1. **High Extensibility**
    - New log types can be added through the implementation of aggregators and of new parsers without the need to modify code that is existing.
    - Implementing new output writers allows for support of new output formats. These formats thus become supportable now.
    - The system handles changing requirements using code that is modified very little
2. **Separation of Concerns**
    - A responsibility that is clearly defined exists for each component.
    - Parsing logic differs from aggregation logic.
    - Data processing is different from output formatting.
3. **Maintainability**
    - Other components are not affected by one's changes.
    - Comprehend as well as adjust code more simply.
    - Because the relations between components are well-defined, then system structure is in fact clear.
4. **Testability**
    - One can test out components in isolation.
    - Mock objects can be used with dependent components.
    - It is easier to achieve high test coverage
5. **Flexibility**
    - Application changes with parse requirements.

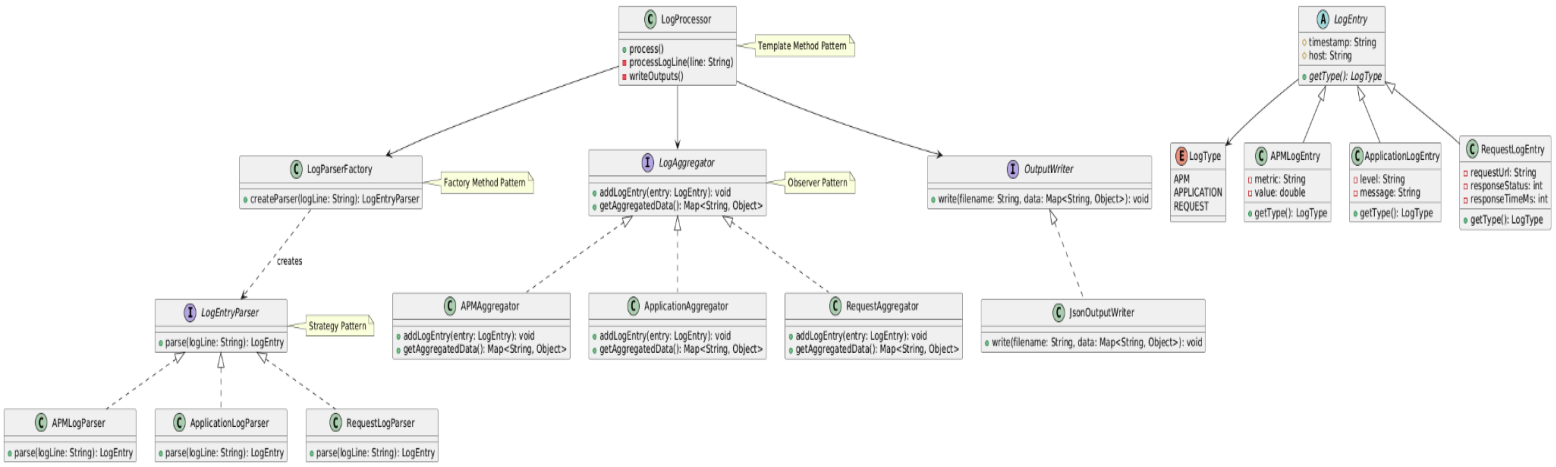
- The core system does not need change at all. Different aggregation strategies can be implemented here.
- Output formats of output can be changed in a way that is independent

Potential Disadvantages

1. **Increased Complexity**
   - More classes as well as interfaces than is a monolithic design simpler.
   - Knowledge of design patterns is needed in order to grasp the system.
   - More files to manage
2. **Learning Curve**
   - For new developers, comprehension of the architecture is in need of time.
   - Documentation becomes more important.
3. **Performance Overhead**
   - The abstraction layers introduce some performance overhead.
   - Methods call through multiple interfaces.
4. **Initial Development Time**
   - Setting this architecture up takes more time at first.
   - Over time the system evolves as benefits appear.

Class Diagram

Here's a class diagram showing the structure of the application and the design patterns used:



Core Components and Their Roles

1. **LogProcessor** (Template Method Pattern)
   - Defines the overall workflow for processing logs
   - Coordinates parsers, aggregators, and output writers

2. **LogEntryParser Interface** (Strategy Pattern)
    - Defines the interface for all parser strategies
    - Allows different parsing approaches for each log type
3. **LogParserFactory** (Factory Method Pattern)
    - Creates appropriate parser instances based on log content
    - Centralizes parser creation logic
4. **Concrete Parsers** (Strategy Pattern implementations)
    - Implement specialized parsing logic for each log type
    - Convert raw log lines into structured log entries
5. **LogEntry Class Hierarchy**
    - Base class with common fields and methods
    - Specialized subclasses for each log type
6. **LogAggregator Interface** (Observer Pattern)
    - Defines methods for collecting and aggregating log data
    - Enables independent data processing for each log type
7. **Concrete Aggregators** (Observer Pattern implementations)
    - Implement specialized aggregation logic for each log type
    - Calculate statistics based on collected log entries
8. **OutputWriter Interface**
    - Abstracts the output mechanism
    - Allows for different output formats
9. **JsonOutputWriter**
    - Implements JSON output formatting
    - Writes aggregated data to files

This design creates a flexible, maintainable, and extensible system that effectively meets the requirements while allowing for future enhancements.