Log Parser System Design Document

1. Describe what problem you're solving.

The problem is to develop a command-line application that processes and analyzes log files containing different types of log entries. Specifically, the application needs to:

1. Parse a log file that contains multiple types of log entries (APM logs, Application logs, and Request logs) in a single file.
2. Classify each log entry into its appropriate category based on its content and structure.
3. Perform specialized aggregations for each log type:
    - For APM logs: Calculate statistical metrics (minimum, median, average, maximum) for various performance indicators.
    - For Application logs: Count logs by severity level (ERROR, WARNING, INFO, DEBUG).
    - For Request logs: Calculate response time statistics (min, percentiles, max) and count HTTP status codes by category (2xx, 4xx, 5xx) for each API route.
4. Output the aggregation results to separate JSON files for each log type.
5. Handle malformed or unrecognized log entries gracefully.
6. Be designed in a way that allows for easy extension to support new log types and file formats in the future.

This problem requires a solution that not only meets the current requirements but also accommodates future changes without significant code modifications.

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

To address this problem effectively, I've implemented two primary design patterns:

Strategy Pattern

The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithm vary independently from clients that use it.

In this application:

- The `LogProcessor` interface defines the strategy for processing different log types.
- Concrete implementations (`APMLogProcessor`, `ApplicationLogProcessor`, `RequestLogProcessor`) provide specialized algorithms for each log type.
- This allows the application to dynamically select the appropriate processing algorithm based on the log entry's format.

Similarly, the `LogAggregator` interface with its concrete implementations follows the Strategy pattern for different aggregation algorithms.

Factory Pattern

The Factory pattern provides an interface for creating objects but allows subclasses to alter the type of objects that will be created.

In this application:

- The `LogProcessorFactory` class encapsulates the logic for determining which processor to use for each log line.
- It examines each log line and returns the appropriate processor instance without exposing the instantiation logic.
- This centralizes the creation logic and decouples the client code from the specific processor implementations.

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

Benefits

1. **Separation of Concerns**: Each log type's processing and aggregation logic is isolated in dedicated classes, making the code more maintainable and easier to understand.

2. **Open/Closed Principle**: The system follows the open/closed principle—it's open for extension (new log types can be added) but closed for modification (existing code doesn't need to change).

3. **Improved Testability**: Components can be tested independently, which simplifies unit testing and increases test coverage.

4. **Flexibility and Extensibility**: Adding support for new log types only requires creating new implementations of the existing interfaces, without modifying the core processing logic.

5. **Code Reusability**: Common functionality is abstracted into base classes and interfaces, reducing code duplication.

6. **Runtime Flexibility**: The application can dynamically select appropriate processors and aggregators at runtime based on the input data.

Drawbacks

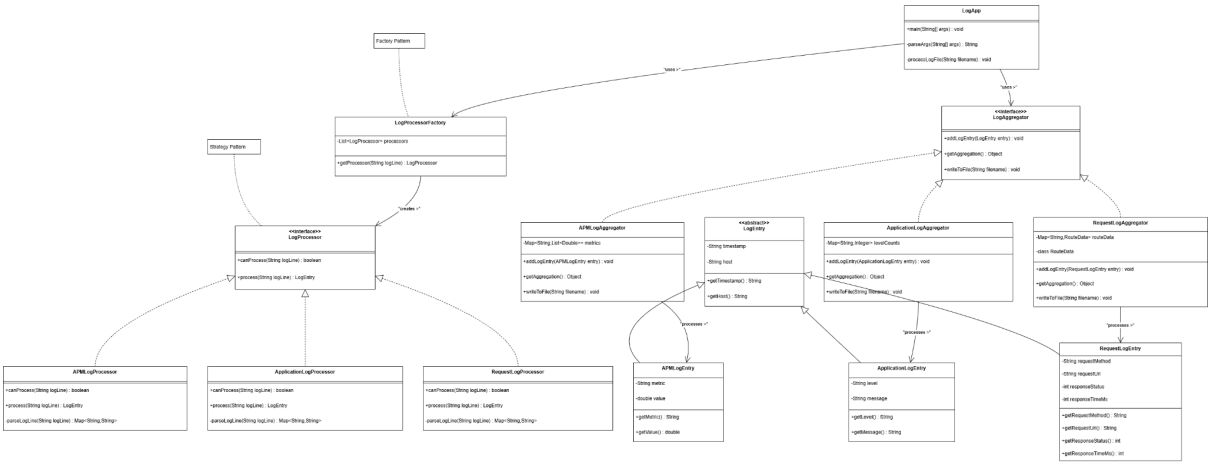
1. **Increased Complexity**: The use of design patterns introduces more classes and interfaces, which can make the system more complex to understand initially.

2. **Performance Overhead**: The additional abstraction layers may introduce a slight performance cost due to the increased number of method calls and object instantiations.

3. **Learning Curve**: Developers who are unfamiliar with these patterns may need time to understand the system architecture.

4. **Potential Over-Engineering**: For simpler applications, this level of abstraction might be unnecessary. However, given the requirement for extensibility, it's justified in this case.

4. Class Diagram



The class diagram illustrates the implementation of both the Strategy and Factory patterns:

- The **Strategy Pattern** is visible in the `LogProcessor` and `LogAggregator` interfaces with their respective concrete implementations.
- The **Factory Pattern** is implemented through the `LogProcessorFactory` class that creates the appropriate processor based on the log content.

The diagram shows the inheritance relationships from abstract classes/interfaces to concrete implementations, as well as the associations between components. This structure enables the application to process different log types with specialized algorithms while maintaining a clean, extensible architecture.