Design Document - 202 Individual Project

Author: Kalyani Chitre (017622917)
Project: Log Processing CLI Application

1. Problem Description

The modern software systems create huge amounts of log files which are a representation of performance metrics, operational events, and API request activity. Logs from different even categories may be mixed in a single file. This results in hard extraction of structured insights.

The command-line Java application that is going to be developed in this project is going to work on a raw log file containing three different log entries:

- APM Logs — system performance metrics
- Application Logs — informational, warning, and error messages
- Request Logs — incoming API requests, response times, and status codes

The application is required to:

- Identify each log type from raw lines
- Parse fields relevant to that type
- Aggregate metrics per specification
- Output results into three separate JSON files: apm.json, application.json, and request.json

It must be capable of safely ignoring and recognizing log lines that have been corrupted and unrecognized. The architecture should also be such that it is easy to add new log types or data sources in the future. The creation of parsing and aggregation components that are modular and extensible will make the solution maintainable, testable, and adaptable to the changes in system requirements.

2. Design Patterns Used

In order to create a design that is both modular and extensible, various software design patterns have been introduced:

2.1 Factory Pattern

The Factory Pattern serves to make a selection at runtime regarding the parser that is to process a log line.

Why This Pattern Is Used

- It is common for every log type to have its own set of identifiers like:
    - metric= for APM logs
    - level= for application logs
    - request_method= for request logs
- The factory not only encapsulates but also effectively manages the decision-making process, thereby avoiding the hard-coding of conditional logic across the system.

Benefits

- New log types can be added easily without the need to alter the core logic.
- The determination of log types is centralized in one place.
- The code is kept cleaner and more maintainable.

Factory Responsibility

- Take in one log line
- Deliver the suitable LogParser implementation
- Deliver null for lines that are either corrupt or unknown

2.2 Strategy Pattern

Each log type has its own specific parsing rules and aggregation behavior. To keep the logic from turning monolithic, a Strategy Pattern is used through the LogParser interface.

Why This Pattern Is Used

- APM logs require extraction of numerical metrics
- Application logs require parsing of the severity (INFO, DEBUG, ERROR, etc.)
- Request logs need URLs, response times, and statuses to be extracted
- Each log type also has a different aggregation logic

Benefits

- Isolation of every parser makes it completely testable
- LogProcessor is simplified as it does not require per-type logic
- The addition of new parsing algorithms is now very easy

Strategies Examples

- APMLogParser
- ApplicationLogParser
- RequestLogParser

2.3 Singleton Pattern

Though not mandatory, the Singleton Pattern has been implemented in the project for the aggregator manager to provide an even state throughout the application.

Why This Pattern Is Used

- Aggregators are responsible for collecting data from all file parts
- Only one instance of every aggregator should be present during the execution
- Avoids unintentional duplication and makes state management easier

Benefits

- Assures the same aggregation behavior to be carried out
- Lowers the chance of bugs arising from many aggregator instances

3. Consequences of Using These Patterns

3.1 Factory Pattern Consequences

Positive

- Increased flexibility: new log types can be added with just a bit of change
- Classification logic is simplified
- Code duplication is diminished

Negative

- Class count rises slightly
- Wrong use can result in complicated hierarchies

Impact

The factory method provides a boost in maintainability and is able to nurture future improvements without the need for a complete re-write of the existing logic, which is also very much in line with project objectives.

3.2 Strategy Pattern Consequences

Positive

- Very clear separation of concerns
- Single components are very testable
- No more huge if/else blocks used in the parser.

Negative

- Demands more planning for the structure and interfaces
- Developers are required to have a good grasp of polymorphism to create the behavior.

Impact

The strategy pattern offers the required flexibility to accommodate new log types in the future, while still maintaining the system's coherence and modularity.

3.3 Singleton Pattern Consequences

Positive

- Only one aggregation state is guaranteed to be present
- Incidental misuse of the aggregators is minimized
- Output generation is made predictable

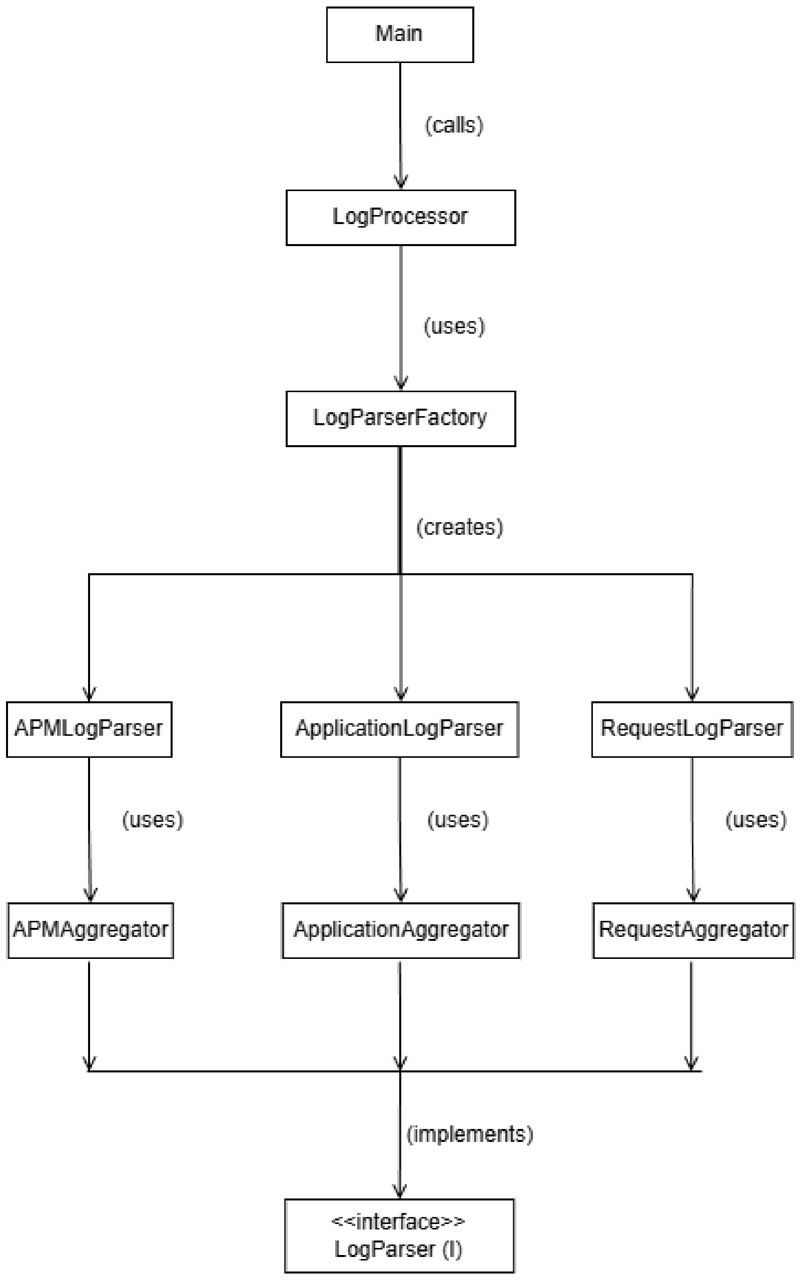
Negative

- If unit tests are not implemented properly, they might become complicated
- If not used properly, a potential global state issue arises

Impact

For the case of this small application, a singleton aggregator manager makes the workflow simpler without generating significant disadvantages.

4. UML Class Diagram



5. Summary

This design puts together the log processing system in a structured and efficient manner using three pivotal design patterns: Factory, Strategy, and Singleton. These empower the application to categorize the logs properly, decode them correctly, and summarize significant metrics not only that but also be completely ready to take on new log types down the road.
When the various functions of parsing, aggregating, and processing are done separately, the system is still flexible, easy to maintain, and scalable.