202 Individual Project:

Part I: Design Outline:

**1. Problem Statement**

We are solving the problem of parsing a unified application log file containing multiple types of logs (APM, Application, Request), classifying them, and performing meaningful aggregations, then exporting each into its respective JSON file. The system must be extensible to support more log types and file formats in the future.

2. **Design Pattern(s) Used**

| **Pattern** | **Purpose** |
| --- | --- |
| **Strategy Pattern** | To encapsulate log parsing and aggregation logic for each log type (APM, Application, Request) separately. |
| **Factory Pattern** | To instantiate appropriate log parser based on the content of each log line. |
| **Template Method Pattern** *(optional)* | To define the skeleton for parsing/aggregating a log type while letting subclasses define specific steps. |
| **Singleton Pattern** | For managing shared utilities like JSON writer or statistics computation service. |

**3. Consequences of Using These Patterns**

* ✅ **Scalability:** Easily add new log types without touching existing code (Open-Closed Principle).
* ✅ **Separation of Concerns:** Each class handles a single responsibility—parsing, aggregating, writing.
* ✅ **Testability:** Individual strategies can be unit-tested with JUnit.
* 🔄 **Slightly More Classes:** More boilerplate due to pattern overhead.
* 🔄 **Learning Curve:** Slight complexity for new developers unfamiliar with Strategy + Factory use.

**✅ 1. Describe what problem you’re solving**

In this project, we address the problem of parsing, classifying, and aggregating mixed-format log entries from a unified .txt log file. In many enterprise applications, logs are generated in large volumes and in heterogeneous formats, making manual analysis error-prone and inefficient. This project automates the extraction of insights from such logs by categorizing them into three types: APM logs (application performance metrics), Application logs (log levels like ERROR, INFO), and Request logs (API request details).

For each category, the application performs structured aggregations:

* APM logs are aggregated to compute statistical summaries (min, max, average, median) for each metric (e.g., CPU usage).
* Application logs are aggregated by log severity levels.
* Request logs are analyzed to compute response time percentiles and categorize HTTP status codes per endpoint.

The application outputs the aggregated results into structured, human-readable JSON files (apm.json, application.json, and request.json) for further analysis or integration with dashboards.

This system is designed to be extensible so that additional log types and formats can be integrated with minimal changes.

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

This application uses a combination of **three design patterns**:

**1. Strategy Pattern**

The Strategy pattern is employed to encapsulate different log parsing behaviors. Each log type (APM, Application, Request) has its own parser class (APMLogParser, ApplicationLogParser, and RequestLogParser), all of which implement a common interface LogParser. This allows each log type to define its own parsing logic independently of others.

**2. Factory Pattern**

The Factory pattern is implemented via the LogParserFactory class. This class determines the appropriate parser strategy based on the structure of a log line. It abstracts the logic of selecting the correct parser, simplifying the code in the main application logic.

**3. Singleton-Like Utility Pattern**

Utility classes such as StatUtils and JsonWriterUtil provide common services like statistical computation and JSON writing. While not singletons in the formal sense, these are stateless, static method–based classes that serve shared functionality across all components.

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

The selected patterns bring multiple benefits and trade-offs to the implementation:

**🔹 Advantages**

* **Open-Closed Principle**: Adding a new log type requires implementing a new parser and updating the factory—existing logic remains untouched.
* **High Cohesion**: Each parser class is focused solely on its parsing task; aggregators handle aggregation, and utilities handle I/O and math logic.
* **Ease of Testing**: Since each component is modular, it can be independently unit tested with high precision.
* **Maintainability**: Future updates (e.g., adding logs like security events or database queries) are simplified by extending the strategy and factory layers.

**🔸 Trade-offs**

* **Increased Class Count**: The use of design patterns introduces more classes and interfaces, which can be overkill for very small-scale projects.
* **Slight Learning Curve**: Developers unfamiliar with these patterns might need time to understand the architecture.
* **Runtime Overhead**: Dynamic dispatch through the factory and strategy adds slight runtime overhead (though negligible in this use case).

Overall, the benefits of extensibility, clarity, and scalability outweigh the minor trade-offs, especially considering the potential growth of the system.