Answer/outline the following:

1. **Describe what problem you're solving.**

The problem being solved involves parsing a log file, extracting specific types of log entries categorized based on specific criteria, and then performing different aggregations based on these logs. The end goal is to write these aggregations into distinct output JSON files, each tailored to its respective log category. By categorizing and aggregating these logs, the program aims to provide comprehensive insights into the performance, operation, and usage of the application, facilitating effective monitoring, analysis, and optimization efforts.

The type of log entries are as follows -

- **Application Performance Metrics:** These logs pertain to various metrics related to the performance of the application, such as CPU, disk, and memory usage. The objective is to analyze these metrics and collect statistics like minimum, median, average, and maximum values.

- **Application Logs:** This category encompasses events related to the operation of the application. It includes errors, warnings, and informational messages that offer insights into the behavior and health of the application. The aim here is to aggregate the count of logs based on severity level.

- **Request Logs:** Request logs provide details about the requests made to the application, including the request type, endpoint, response time, and status code. Aggregations for request logs involve:
  - Calculating metrics like minimum, maximum, 50th percentile, 90th percentile, 95th percentile, and 99th percentile response times per API route.
  - Counting the number of requests categorized by HTTP status code (e.g., 2xx, 4xx, 5xx) per API route.

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

Strategy pattern is an effective solution for this problem as the behaviour of the log processing varies based on different log types. Each strategy class encapsulates its own log processing algorithm details, reducing code duplication and improving maintainability. New log processing strategies can be added without modifying existing code, promoting flexibility and scalability.

<u>**Implementation details -**</u>

- Define an interface - "**LogProcessor**" that declares methods for processing log types. This interface will be implemented by concrete strategy classes.
- Implement concrete strategy classes (**ApmLogProcessor**, **AppLogProcessor**, **RequestLogProcessor**) that encapsulate the algorithms for processing each type of log.
- Each concrete strategy class will implement methods specific to its type of log processing. For example, processLogs() might calculate statistics like minimum, median, average, and maximum values for CPU, disk, and memory usage metrics.

- The context class (**LogAnalyzer**) will maintain a reference to the current log processing strategy. It will delegate log processing tasks to the strategy object without knowing the specifics of how each type of log is processed.
- Clients of the LogProcessor class can dynamically switch between log processing strategies based on the type of logs being processed.

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

Utilizing the Strategy pattern in the log processing system has several consequences, both advantageous and potentially limiting. Here are the key consequences:

<u>**Advantages:**</u>

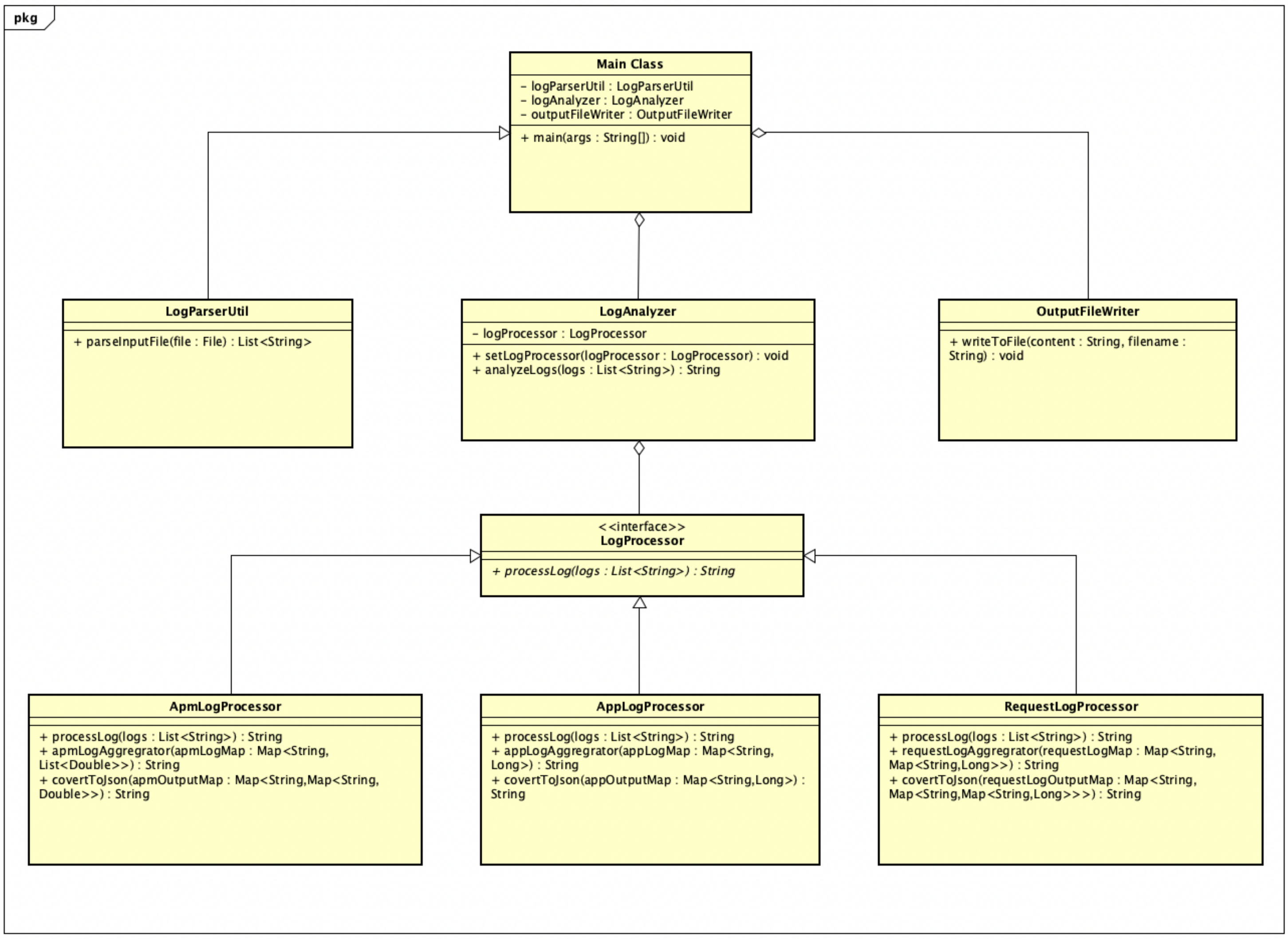
- **Flexibility:** The Strategy pattern promotes flexibility by allowing log processing algorithms to vary independently of the clients that use them. New log processing strategies can be added or existing ones modified without affecting the clients, providing flexibility to adapt to changing requirements.
- **Modularity:** Each log processing algorithm is encapsulated within its own strategy class, promoting modularity and separation of concerns. This enhances code organization, readability, and maintainability.
- **Reusability:** Strategies can be reused across different parts of the system or even in different projects, enhancing code reuse and reducing duplication.
- **Testability:** Since each strategy encapsulates a specific algorithm, it becomes easier to test individual strategies in isolation, leading to more effective unit testing.

<u>**Limitations:**</u>

Increased Complexity: Introducing multiple strategies and the associated context class can increase the complexity of the system. Developers need to manage the interactions between the context and the strategies, potentially leading to increased cognitive load.

- <u>**Runtime Overhead:**</u> The Strategy pattern involves dynamic composition, allowing strategies to be changed at runtime. While this provides flexibility, it can introduce runtime overhead due to the need for dynamic dispatch and method calls.
- <u>**Potential Overdesign:**</u> In some cases, applying the Strategy pattern might be overkill if the variability in behavior is limited or unlikely to change. Using the Strategy pattern in such scenarios could lead to unnecessary complexity and overhead.
- <u>**Increased Number of Classes:**</u> Introducing a separate strategy class for each variation in behavior can lead to a proliferation of classes, which might increase the overall codebase size and complexity.

4. **Create a class diagram – showing your classes and the Chosen design pattern**


pkg
Main Class
- logParserUtil : LogParserUtil
- logAnalyzer : LogAnalyzer
- outputFileWriter : OutputFileWriter
+ main(args : String[]) : void
LogParserUtil
+ parseInputFile(file : File) : List<String>
LogAnalyzer
- logProcessor : LogProcessor
+ setLogProcessor(logProcessor : LogProcessor) : void
+ analyzeLogs(logs : List<String>) : String
OutputFileWriter
+ writeToFile(content : String, filename : String) : void
<<interface>>
LogProcessor
+ processLog(logs : List<String>) : String
ApmLogProcessor
+ processLog(logs : List<String>) : String
+ apmLogAggregrator(apmLogMap : Map<String, List<Double>>) : String
+ covertToJson(apmOutputMap : Map<String,Map<String, Double>>) : String
AppLogProcessor
+ processLog(logs : List<String>) : String
+ appLogAggregrator(appLogMap : Map<String, Long>) : String
+ covertToJson(appOutputMap : Map<String,Long>) : String
RequestLogProcessor
+ processLog(logs : List<String>) : String
+ requestLogAggregrator(requestLogMap : Map<String, Map<String,Long>>) : String
+ covertToJson(requestLogOutputMap : Map<String, Map<String,Map<String,Long>>>) : String