Part I (25 points)

Answer/outline the following:

1. Describe what problem you're solving.

This project solves the problem of automated log classification and aggregation in software systems. In real-world applications, log files often contain a mix of performance metrics, application events, and HTTP requests. Manually parsing and analyzing such logs is inefficient, error-prone, and time-consuming.

This project:

- Parses a mixed log file (APM, application, request logs)
- Classify each line based on its structure and contents
- Aggregate the data for analysis (e.g., computing averages, counts, percentiles)
- Outputs results in structured .json format

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

The Factory Pattern was used to create different types of log aggregators (ApmAggregator, ApplicationAggregator, RequestAggregator) based on the type of log entry being processed — without hardcoding class instantiations throughout the code. Strategy pattern is also utilized to interchange between aggregator classes.

When processing each line of a log file, the log is classified as either apm, application, or request. Each type has a different aggregation strategy (e.g., statistical percentiles vs. log level counts). Creating aggregators directly would tightly couple main logic. Instead the LogAggregatorFactory takes the log type as input and returns correct aggregator object

```
ILogAggregator appAgg = factory.createAggregator("application");
```

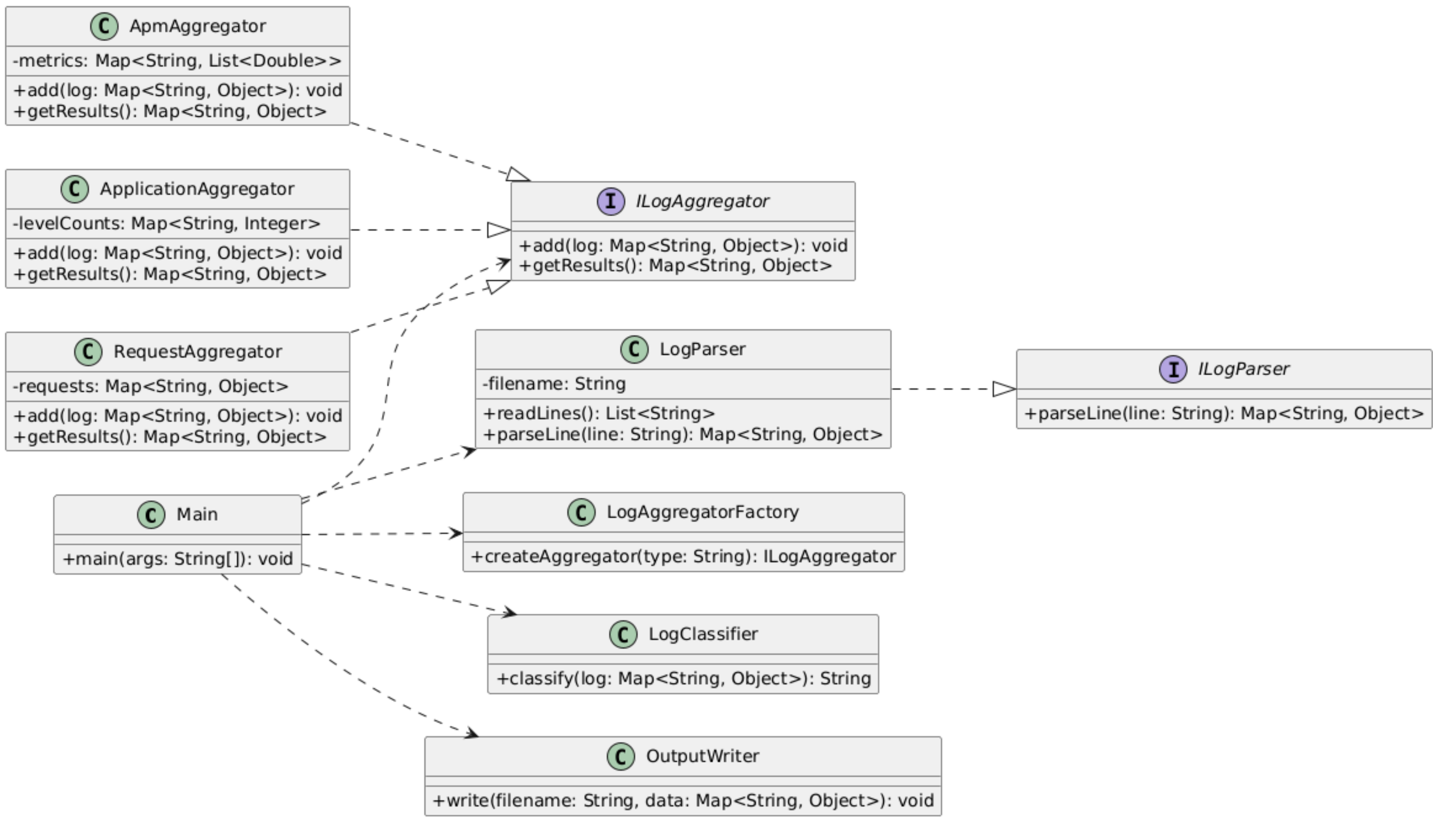
Result: Main class remains decoupled from implementation details while still producing right behavior.
Benefits of Factory Pattern:

- Creates right Aggregator based on log type and behavior
- Encapsulates all object creation logic, avoids if/else/switch statements in main logic. This violates Open/Closed principles as you have to modify this logic every time you add a new type.
- Can use ILogAggregator to update factories and add new types easily.
- Promotes clean and reusable code, creation logic is in one place.

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

Overhead, developers need to trace through factory logic to understand object instantiation.Unit testing increases in complexity as mocks may be needed to isolate behaviors.

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


ApmAggregator
-metrics: Map<String, List<Double>>
+add(log: Map<String, Object>): void
+getResults(): Map<String, Object>
ApplicationAggregator
-levelCounts: Map<String, Integer>
+add(log: Map<String, Object>): void
+getResults(): Map<String, Object>
RequestAggregator
-requests: Map<String, Object>
+add(log: Map<String, Object>): void
+getResults(): Map<String, Object>
ILogAggregator
+add(log: Map<String, Object>): void
+getResults(): Map<String, Object>
LogParser
-filename: String
+readLines(): List<String>
+parseLine(line: String): Map<String, Object>
ILogParser
+parseLine(line: String): Map<String, Object>
Main
+main(args: String[]): void
LogAggregatorFactory
+createAggregator(type: String): ILogAggregator
LogClassifier
+classify(log: Map<String, Object>): String
OutputWriter
+write(filename: String, data: Map<String, Object>): void