CMPE-202 Individual Project Deliverables

Part I (25 points)

Answer/outline the following:

- Describe what problem you're solving.
- What design pattern(s) will be used to solve this?
- Describe the consequences of using this/these pattern(s).
- Create a class diagram - showing your classes and the Chosen design pattern

---

1. Problem Description

Problem Statement: To build a command-line Java application that parses a log file containing various types of logs (APM, Application, Request), classifies each log entry, and writes aggregated results for each log type into separate JSON files. The application must be extensible to support new log types and file formats in the future.

2. Design Patterns Used

1. Factory Method Pattern

How it's used:

- The LogEntryFactory class contains a method (createLogEntry) that examines each log line and decides which subclass of LogEntry to instantiate (APMLogEntry, ApplicationLogEntry, or RequestLogEntry).

Purpose:

- To encapsulate the logic for creating different types of log entry objects based on the content of each log line.
- To centralize and isolate the object creation process, so the rest of the code doesn't need to know the details of how to create each log entry type.

Why it's better:

- **Extensibility:** If you want to add a new log type in the future, you only need to add a new subclass and update the factory, not the rest of your code.
- **Separation of concerns:** Parsing and object creation are kept separate from business logic.
- **Cleaner code:** The main parsing loop doesn't need a big if-else or switch statement for every log type.

Cons:

- **Slightly More Complex:** Introduces an extra layer (the factory) which can make the codebase a bit more complex for small/simple projects.
- **Factory Maintenance:** The factory class must be updated whenever a new log type is added, which can become a maintenance point if not managed well.

2. Strategy Pattern

How it's used:

- Each log type has its own aggregator class (APMAggregator, ApplicationAggregator, RequestAggregator) that implements the LogAggregator interface.
- The main parser uses these aggregators interchangeably, depending on the log entry type.

Purpose:

- To encapsulate different aggregation algorithms for different log types.
- To allow the main parsing logic to use any aggregator without knowing its internal details.

Why it's better:

- **Open/Closed Principle:** You can add new aggregation strategies (for new log types) without modifying existing code.
- **Flexibility:** You can change how aggregation is done for a log type by swapping out the strategy.
- **Testability:** Each aggregator can be tested independently.

Cons:

- **More Classes:** Increases the number of classes in the project, which can make navigation harder if not organized well.
- **Initial Setup:** Requires more upfront design and setup compared to a simple if-else approach.

3. Single Responsibility Principle (SRP) & Open/Closed Principle (OCP)

How it's used:

- Each class has a single responsibility: parsing, aggregating, writing JSON, etc.
- The system is open for extension (add new log types/aggregators) but closed for modification (existing code doesn't need to change).

Purpose:

- To make the codebase easy to maintain and extend.
- To reduce the risk of bugs when adding new features.

Why it's better:

- **Maintainability:** Changes in one part of the system don't affect others.
- **Scalability:** Easy to add new log types or output formats.
- **Robustness:** Less chance of breaking existing features when adding new ones.

Cons:

- **More Files/Classes:** Following SRP and OCP can lead to more files and classes, which may seem overwhelming for very small projects.
- **Learning Curve:** Developers unfamiliar with these principles may need time to understand the structure.

Part II (75 points)

Implement an application (Java code and JUnit tests) for Part 1. Output should contain the details specified in the project description.

This section of the project focused on building a modular Java command-line tool that processes a mixed log file and produces structured JSON aggregations for three log categories: APM logs, Application logs, and Request logs.

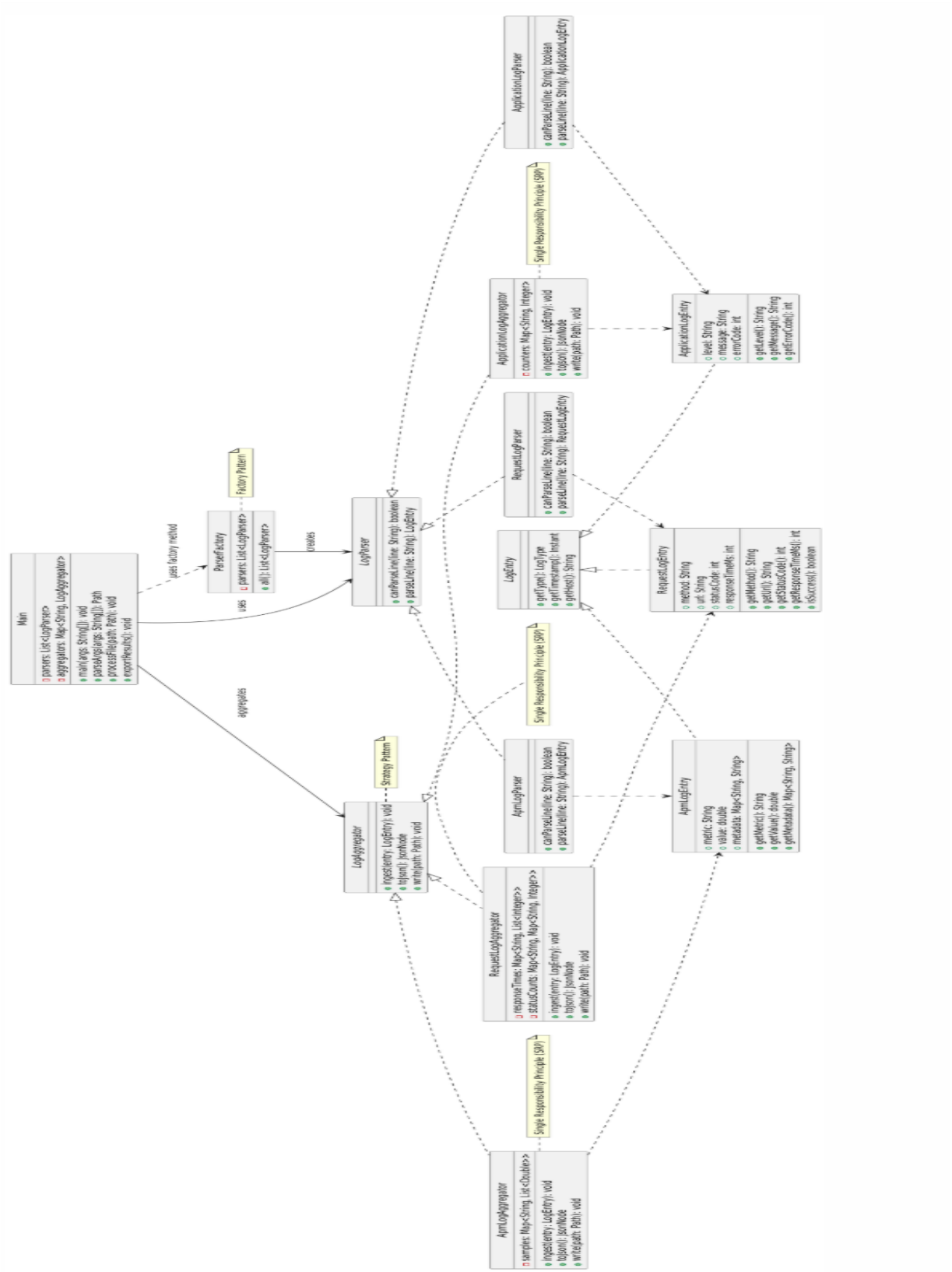
Output Files

- **apm.json** – Contains aggregated APM metrics.
- **application.json** – Contains counts of log entries by severity level.
- **request.json** – Contains response time statistics and status code distributions for each API route.

Testing

- **Unit tests** were written using **JUnit 5**, with a dedicated test class for each aggregator:
- ApmAggregatorTest checks the correct calculation of minimum, median, average, and maximum metric values.
- ApplicationAggregatorTest verifies the correct counting of log severities such as INFO, ERROR, and DEBUG.
- RequestAggregatorTest ensures accurate computation of response time percentiles and HTTP status code distributions per route.
- Each test provides controlled input to the aggregator and uses assertions to compare the generated JSON output with expected results.

UML Diagram:


Main
parsers: List<LogParser>
aggregators: Map<String, LogAggregator>
+ main(args: String[]): void
+ parseArgs(args: String[]): Path
+ processFile(path: Path): void
+ exportResults(): void
ParserFactory
- parsers: List<LogParser>
+ all(): List<LogParser>
Factory Pattern
uses factory method
uses
creates
aggregates
LogParser
+ canParse(line: String): boolean
+ parse(line: String): LogEntry
ApplicationLogParser
+ canParse(line: String): boolean
+ parse(line: String): ApplicationLogEntry
RequestLogParser
+ canParse(line: String): boolean
+ parse(line: String): RequestLogEntry
AgmLogParser
+ canParse(line: String): boolean
+ parse(line: String): AgmLogEntry
LogEntry
+ getType(): LogType
+ getTimestamp(): Instant
+ getHost(): String
ApplicationLogEntry
○ level: String
○ message: String
○ errorCode: int
+ getLevel(): String
+ getMessage(): String
+ getErrorCode(): int
RequestLogEntry
○ method: String
○ url: String
○ statusCode: int
○ responseTimeMs: int
+ getMethod(): String
+ getUrl(): String
+ getStatusCode(): int
+ getResponseTimeMs(): int
+ isSuccess(): boolean
AgmLogEntry
○ metric: String
○ value: double
○ metadata: Map<String, String>
+ getMetric(): String
+ getValue(): double
+ getMetadata(k): Map<String, String>
Single Responsibility Principle (SRP)
Single Responsibility Principle (SRP)
Single Responsibility Principle (SRP)
ApplicationLogAggregator
- counters: Map<String, Integer>
+ ingest(entry: LogEntry): void
+ toJson(): JsonNode
+ writeJson(path: Path): void
RequestLogAggregator
- responseTimes: Map<String, List<Integer>>
- statusCounts: Map<String, Map<String, Integer>>
+ ingest(entry: LogEntry): void
+ toJson(): JsonNode
+ writeJson(path: Path): void
AgmLogAggregator
- samples: Map<String, List<Double>>
+ ingest(entry: LogEntry): void
+ toJson(): JsonNode
+ writeJson(path: Path): void
LogAggregator
+ ingest(entry: LogEntry): void
+ toJson(): JsonNode
+ writeJson(path: Path): void
Strategy Pattern