Name: Harshavardhan Valmiki
SJSU ID: 017442984

Part I: **25pts**

1) Describe the Problem You're Solving

The task at hand is to create a command-line tool capable of parsing a text-based log file efficiently. Each line in the log file represents a separate log entry, and the objective is to classify these logs into three distinct types: APM (Application Performance Metrics) Logs, Application Logs, and Request Logs. Each category necessitates specific aggregation techniques to derive insights such as minimum, median, average, and maximum values for metrics, counts based on severity for application events, and statistics regarding response times for web requests. The solution should be resilient enough to handle corrupted lines gracefully and flexible to accommodate potential additions of new log types and alterations in file formats in the future.

2) What Design Pattern(s) Will Be Used to Solve This?

The combination of the Strategy and Factory design patterns presents an effective solution for the described problem:

Strategy Pattern: This pattern is ideal for defining a family of algorithms, encapsulating each, and allowing them to be interchangeable. In the context of your application, different log types (APM, Application, Request) can be processed using distinct strategies. Each log type can have its own algorithm for parsing and aggregating data, facilitating easy swapping of strategies without impacting the overall system. This promotes code flexibility and reusability.

Factory Pattern: Employing this pattern to create instances of different parser classes based on the log type is beneficial. A factory can manage the instantiation logic, determining which log parsing strategy to employ based on the input log line. This approach ensures scalability to support new log types seamlessly, adhering to the open/closed principle.

By leveraging these patterns together, you can establish a robust design capable of dynamically handling various log types while maintaining a clear separation of concerns within the application.

Name: Harshavardhan Valmiki
SJSU ID: 017442984

3) Describe the Consequences of Using These Patterns

Utilizing the Strategy and Factory design patterns within your application will yield several advantageous outcomes:

Enhanced Flexibility in Log Type Expansion:

Strategy Pattern: By encapsulating log processing algorithms within distinct strategy classes, the integration of new log types is simplified. The addition of new strategies without altering existing code enables a highly adaptable system.

Factory Pattern: This pattern allows the application to remain agnostic to the specific classes required for each log type. As new log types are introduced, updating the factory to include new classes maintains a loosely coupled architecture.

Streamlined Maintenance:

Both patterns advocate for component decoupling within the application. This separation minimizes dependencies, simplifying debugging and upkeep. Changes in one part of the system, such as adding new log types or altering aggregation logic, are less likely to impact others.

Promotion of Reusability:

The Strategy pattern facilitates the reuse of strategies across different application segments if necessary. For instance, similar aggregation calculations required across various log types can employ the same strategy.

Facilitation of Scalability:

With the Factory pattern managing object creation and the Strategy pattern governing object behavior, scaling the application to handle more intricate scenarios or increased log volumes becomes more manageable.

Consideration of Complexity:

While these patterns offer substantial advantages, they introduce a degree of complexity into the application's architecture. Familiarizing new developers or maintainers with the design patterns may be necessary for effective engagement with the application.

Acknowledgment of Overhead:

Implementing multiple design patterns may entail increased runtime and developmental overhead. This is due to the involvement of more classes and interfaces compared to a

Name: Harshavardhan Valmiki
SJSU ID: 017442984

4) Class Diagram:

Below are the essential elements for your class diagram:

1. LogParser (Interface)
   - Functions: parse_log(log_line: str)
2. APMLogParser (Class, implements LogParser)
   - Functions: parse_log(log_line: str)
3. ApplicationLogParser (Class, implements LogParser)
   - Functions: parse_log(log_line: str)
4. RequestLogParser (Class, implements LogParser)
   - Functions: parse_log(log_line: str)
5. LogParserFactory (Class)
   - Functions: get_parser(log_line: str) -> LogParser
6. LogAggregator (Class)
   - Properties: parsers: dict
   - Functions: add_parser(log_type: str, parser: LogParser), aggregate_logs(log_file: str)
7. LogFileManager (Class)
   - Properties: file_path: str
   - Functions: read_log_file() -> str

Relationships

- LogParserFactory contains a method get_parser, which, based on the log line, creates and returns an object of either APMLogParser, ApplicationLogParser, or RequestLogParser.
- LogAggregator utilizes instances of LogParser to manage the parsing of log lines based on their type. It includes a dictionary (parsers) that maps log types to their corresponding parsers.
- LogFileManager is responsible for accessing the log file and supplying log lines to the LogAggregator.

Name: Harshavardhan Valmiki
SJSU ID: 017442984