202 - Individual Project

Problem Description

The problem being solved involves parsing and processing diverse types of log data from an application to extract meaningful statistics and insights. The data comes in various formats, each representing different aspects of application behavior, such as performance metrics, application errors, and HTTP request details. The challenge is to efficiently parse these different log types and aggregate relevant information without hard-coding specific behaviors, ensuring the system can easily adapt to new log formats in the future.

Design Patterns Used

1. Strategy Pattern
   - **Usage**: This pattern is used to define a family of algorithms (parsers in this case), encapsulate each one, and make them interchangeable. The strategy pattern lets the algorithm vary independently from clients that use it. In the context of log parsing, each log type (APM, Application, Request) has a corresponding parser that implements a common interface.

2. Factory Method Pattern
   - **Usage**: This pattern is used to handle the creation of objects without specifying the exact class of object that will be created. This is achieved by defining an interface for creating an object, but letting subclasses decide which class to instantiate. The LogParserFactory uses this pattern to instantiate the appropriate parser based on the log type detected.

Consequences of Using These Patterns

1. Strategy Pattern
   - **Pros**:
     - **Flexibility**: It provides the flexibility to switch between different parsing algorithms at runtime depending on the log entry type.
     - **Scalability**: New parsing strategies can be added without altering the code that uses the parsers.
   - **Cons**:
     - **Increased Complexity**: It can complicate the codebase by requiring multiple classes to implement the same interface.

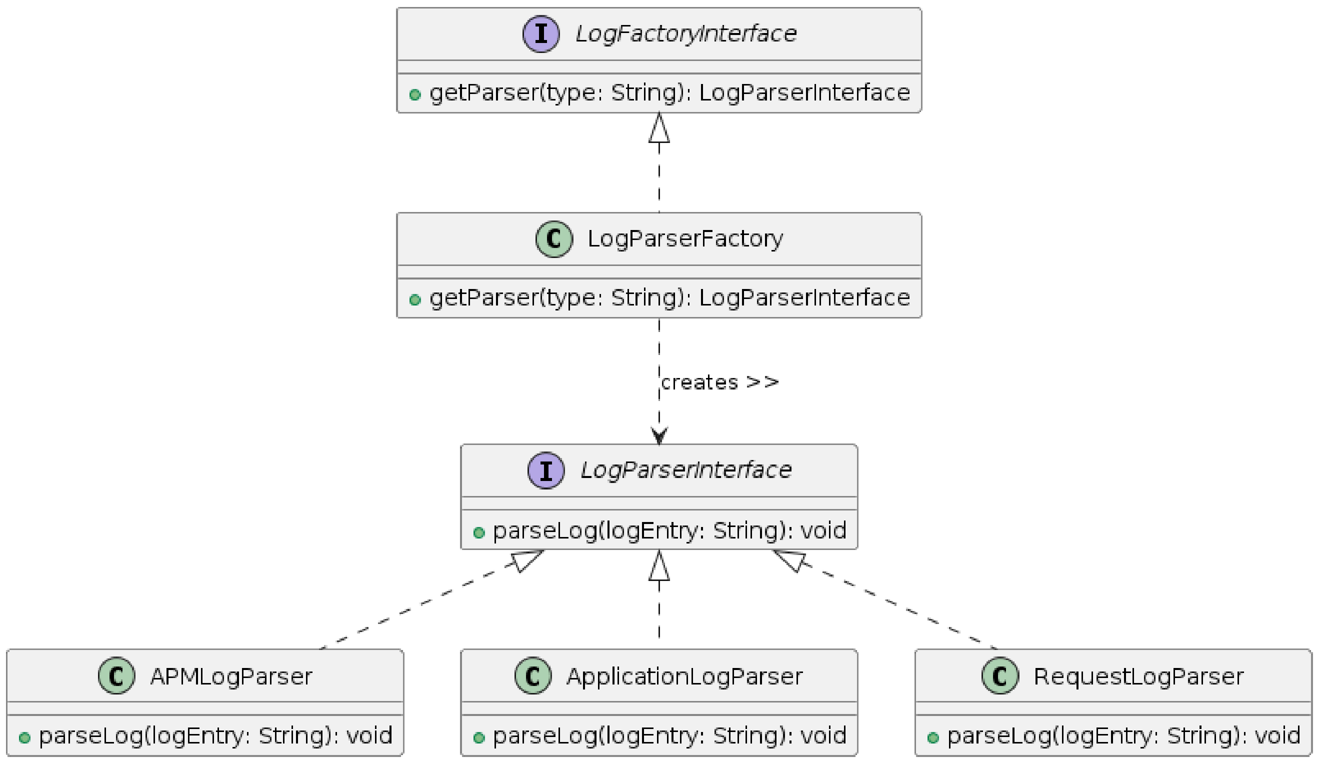
        - **Overhead**: If not well-managed, can introduce runtime overhead due to dynamic invocation of methods.

2. Factory Method Pattern

- **Pros**:
    - **Encapsulation of Creation Logic**: It encapsulates the instantiation logic into a single location, simplifying system maintenance.
    - **Extensibility**: Adding new parsers as the system evolves becomes straightforward.
- **Cons**:
    - **Subclassing**: Every new parser type requires a new subclass, which might increase the number of classes in the system unnecessarily.

Class Diagram

Below is a simple class diagram illustrating the use of the Strategy and Factory Method patterns in the context of the log parsing application:



I LogFactoryInterface
getParser(type: String): LogParserInterface
C LogParserFactory
getParser(type: String): LogParserInterface
creates >>
I LogParserInterface
parseLog(logEntry: String): void
C APMLogParser
parseLog(logEntry: String): void
C ApplicationLogParser
parseLog(logEntry: String): void
C RequestLogParser
parseLog(logEntry: String): void

Explanation of the Diagram

- **LogParserInterface**: This is the strategy interface. All specific parsers implement this interface, ensuring they can be used interchangeably by the rest of the application.
- **LogFactoryInterface and LogParserFactory**: These represent the factory method pattern. LogParserFactory decides which parser to instantiate based on the input log type.
- **APMLogParser, ApplicationLogParser, RequestLogParser**: These are concrete implementations of LogParserInterface, each tailored to parse a specific type of log data.

This setup ensures that the log parsing module is robust, flexible, and easy to extend with minimal changes to existing code.