Advanced Java Application: Multi-threaded File Processor with Streams and Design Patterns

Problem Statement:

Create a multi-threaded Java application that processes large files (CSV format) containing user information and performs data transformations. The application should use streams for efficient data handling, and it should implement a singleton pattern for centralized logging and thread pool management.

Approach:

1. Multi-threading: Use Java's ExecutorService for managing threads and optimizing the performance of file processing.
2. Streams: Utilize Java Streams for processing the file content efficiently.
3. Design Patterns: Implement Singleton for logging and Factory Pattern for creating different processors based on user input.
4. File Handling: Use NIO (java.nio.file) for better performance with large files.
5. Performance Optimization: The application should handle large files and process them efficiently without running out of memory.

Java Code Sample:

```
import java.io.*;
import java.nio.file.*;
import java.util.*;
import java.util.concurrent.*;
import java.util.stream.*;
import java.util.logging.*;
import java.text.SimpleDateFormat;

// Singleton Logger for centralized logging
class LoggerSingleton {
    private static Logger logger;

    private LoggerSingleton() {}
```

```
    public static Logger getLogger() {
        if (logger == null) {
            logger = Logger.getLogger(LoggerSingleton.class.getName());
            try {
                FileHandler fileHandler = new FileHandler("application.log", true);
                fileHandler.setFormatter(new SimpleFormatter());
                logger.addHandler(fileHandler);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return logger;
    }
}

// Abstract Processor
interface FileProcessor {
    void processLine(String line);
}

// CSV Processor - Concrete Implementation of FileProcessor
class CSVProcessor implements FileProcessor {
    @Override
    public void processLine(String line) {
        // Split the line into fields (assuming CSV format)
        String[] fields = line.split(",");

        // Simulate data transformation
        String userId = fields[0];
        String userName = fields[1].toUpperCase(); // Transform username to uppercase
        String email = fields[2].toLowerCase();  // Transform email to lowercase

        LoggerSingleton.getLogger().info("Processed user: " + userId + ", " + userName + ",
" + email);
    }
}

// Factory for creating different file processors
class FileProcessorFactory {
    public static FileProcessor createProcessor(String fileType) {
```

```
        if ("csv".equalsIgnoreCase(fileType)) {
            return new CSVProcessor();
        }
        // Add more file types as needed (e.g., XML, JSON processors)
        throw new UnsupportedOperationException("Unsupported file type: " + fileType);
    }
}

// Threaded File Processor Class
class ThreadedFileProcessor {
    private ExecutorService executorService;

    public ThreadedFileProcessor(int numberOfThreads) {
        executorService = Executors.newFixedThreadPool(numberOfThreads);
    }

    // Method to process a file
    public void processFile(Path filePath, String fileType) {
        try (Stream<String> lines = Files.lines(filePath)) {
            FileProcessor processor = FileProcessorFactory.createProcessor(fileType);

            // Process lines in parallel using a thread pool
            lines.parallel().forEach(line -> {
                processor.processLine(line);
            });
        } catch (IOException e) {
            LoggerSingleton.getLogger().severe("Error reading the file: " + e.getMessage());
        }
    }

    // Shutdown the executor service
    public void shutdown() {
        executorService.shutdown();
        try {
            if (!executorService.awaitTermination(60, TimeUnit.SECONDS)) {
                executorService.shutdownNow();
            }
        } catch (InterruptedException e) {
            executorService.shutdownNow();
        }
```

```
    }
}

// Main Application to initiate processing
public class FileProcessingApp {
    public static void main(String[] args) {
        // Initialize the logger
        Logger logger = LoggerSingleton.getLogger();

        // Get the file path and file type from command line arguments
        if (args.length < 2) {
            logger.severe("Usage: java FileProcessingApp <file-path> <file-type>");
            System.exit(1);
        }

        Path filePath = Paths.get(args[0]);
        String fileType = args[1].toLowerCase();

        // Check if the file exists
        if (!Files.exists(filePath)) {
            logger.severe("File not found: " + filePath.toString());
            System.exit(1);
        }

        // Start the file processing with a thread pool
        int numberOfThreads = Runtime.getRuntime().availableProcessors(); // Use all
available processors
        ThreadedFileProcessor processor = new ThreadedFileProcessor(numberOfThreads);

        logger.info("Starting file processing...");
        long startTime = System.currentTimeMillis();

        // Process the file
        processor.processFile(filePath, fileType);

        // Measure the time taken to process the file
        long endTime = System.currentTimeMillis();
        logger.info("File processed in " + (endTime - startTime) + " milliseconds.");

        // Shutdown the processor
```

```
        processor.shutdown();
    }
}
```

Code Explanation:

1. LoggerSingleton Class: This is a Singleton Pattern implementation for centralized logging. It ensures that only one logger instance is used across the application. It also writes logs to a file (application.log).

2. FileProcessor Interface and CSVProcessor Class: The FileProcessor interface defines the contract for processing file lines. The CSVProcessor class implements this interface and handles CSV line processing. It mimics some basic data transform, like making usernames uppercase and emails lowercase.

3. FileProcessorFactory Class: It is implemented using Factory Pattern to create a suitable file processor based on the type (CSV, XML) of file. Thanks to this pattern, it will be simple to add support for new file formats in the future.

4. ThreadedFileProcessor Class: In this code, we demonstrate a multi-threaded class that processes the input file using an ExecutorService which reads each line from the file in parallel using Java Streams.It reads large files efficiently by streaming the content and utilizing multiple threads for processing.

5. FileProcessingApp Class: This is the main entry point of the application. It takes a file path and type from command-line arguments, validates the input, and starts the file processing using a thread pool. It also logs the time taken to process the file.