Overview of the `FileOutputStrategy` Class

Package: `com.cardio_generator.outputs`

Purpose:

The `FileOutputStrategy` class is responsible for outputting patient health data to text files in the cardiovascular data simulator. It implements the `OutputStrategy` interface, writing data such as ECG, blood pressure, blood saturation, blood levels, or alerts to separate files based on data type (label) within a specified base directory. The class ensures thread-safe file access and directory creation, providing a persistent storage mechanism for simulation data suitable for analysis or logging.

Role in the Project:

The `FileOutputStrategy` is a key component of the simulator's output subsystem, alongside `ConsoleOutputStrategy`, `WebSocketOutputStrategy`, and `TcpOutputStrategy`. It is used by data generators (e.g., `ECGDataGenerator`, `BloodPressureDataGenerator`) via the `HealthDataSimulator` when file output is selected (e.g., via the `--output file:<directory>` command-line argument). In the context of Task 7, it persists alerts generated by `AlertGenerator`, which may be structured as `Alert` objects (from `com.alerts`) and processed by `AlertFactory` or `AlertDecorator`, enabling offline analysis of alert events. The class supports scenarios requiring durable storage of simulation data, such as debugging, auditing, or integration with external systems.

Technical Characteristics:

- The class implements the `OutputStrategy` interface, ensuring compliance with the simulator's output strategy pattern.

- It uses a `ConcurrentHashMap` for thread-safe mapping of data labels to file paths, optimizing file access.

- It employs Java NIO (`Files`, `Paths`) for robust file and directory operations, with automatic directory creation and file appending.

- The class is designed for concurrent use, handling data output from multiple threads in `HealthDataSimulator`'s `ScheduledExecutorService`, with comprehensive error handling for I/O operations.

Internal Components and Their Purposes

The `FileOutputStrategy` class consists of fields, a constructor, and a single method. Each component is described below, including its purpose, technical details, and role in the class's functionality.

1. Fields:

- `private String baseDirectory`

- Purpose: Stores the path to the base directory where output files are written, serving as the root for all data files.

- Technical Details:

- Type: `String`, representing a filesystem path (e.g., "/path/to/output").

- Private access ensures encapsulation, set only via the constructor.

- Used in the `output` method to construct file paths and create the directory if it does not exist.

- Role: Defines the storage location for output files, enabling the class to organize data by label in a user-specified directory.

- `public final ConcurrentHashMap<String, String> fileMap = new ConcurrentHashMap<>()`

- Purpose: Maintains a thread-safe mapping of data labels (e.g., "ECG", "Alert") to their corresponding file paths (e.g., "/path/to/output/ECG.txt"), caching paths for efficient access.

- Technical Details:

- Type: `ConcurrentHashMap<String, String>`, a thread-safe hash map optimized for concurrent access.

- Public and final, allowing read-only access to the map (though not typically needed externally) and ensuring the map reference is immutable.

- Initialized empty, populated dynamically in the `output` method using `computeIfAbsent`.

- Keys are labels (e.g., "BloodPressure"), and values are absolute file paths (e.g., "/path/to/output/BloodPressure.txt").

- Role: Optimizes file access by caching label-to-file mappings, ensuring thread-safe and efficient resolution of file paths in concurrent environments.

2. Constructor:

- `public FileOutputStrategy(String baseDirectory)`

- Purpose: Initializes a `FileOutputStrategy` instance with the specified base directory for storing output files.

- Technical Details:

- Parameter: `baseDirectory` (`String`), the path to the directory where files will be written.

- Sets the `baseDirectory` field to the provided value.

- Public access allows instantiation by `HealthDataSimulator` or other components when file output is configured.

- Does not validate `baseDirectory` (e.g., for empty strings or invalid paths), relying on subsequent file operations to handle errors.

- The `fileMap` is implicitly initialized as an empty `ConcurrentHashMap` during instance creation.

- Role: Configures the output strategy with the user-specified directory, preparing it for file-based data output.

3. Method:

- `public void output(int patientId, long timestamp, String label, String data)`

- Purpose: Outputs patient health data to a text file corresponding to the specified label, appending the data in a formatted string: `Patient ID: <patientId>, Timestamp: <timestamp>, Label: <label>, Data: <data>`. Ensures thread-safe file access and creates the base directory if needed.

- Technical Details:

  - Parameters:

    - `patientId` (`int`), the unique identifier of the patient associated with the data.

    - `timestamp` (`long`), the time of data generation (milliseconds since epoch).

    - `label` (`String`), the type of data (e.g., "ECG", "BloodPressure", "Alert").

    - `data` (`String`), the data value (e.g., "0.65", "120/80", "triggered").

  - Return Type: `void`, as the method's effect is writing to a file.

  - Overrides the `output` method from the `OutputStrategy` interface, ensuring compliance with the interface's contract.

  - Logic:

    - Creates the base directory (`Files.createDirectories(Paths.get(baseDirectory))`) if it does not exist, handling `IOException` by printing an error to `System.err` and returning early.

    - Resolves the file path for the label using `fileMap.computeIfAbsent`, generating a path like `<baseDirectory>/<label>.txt` if not already cached.

    - Opens the file in append mode (`StandardOpenOption.APPEND`) with `Files.newBufferedWriter`, creating it if it does not exist (`StandardOpenOption.CREATE`).

    - Uses a `PrintWriter` to write the formatted string: `Patient ID: %d, Timestamp: %d, Label: %s, Data: %s%n`.

    - Closes the file automatically via try-with-resources, ensuring proper resource management.

    - Catches `Exception` (e.g., `IOException` for file write failures), printing an error message to `System.err` with the file path and exception details.

  - Thread Safety: Uses `ConcurrentHashMap` for safe path resolution and relies on `Files.newBufferedWriter` with append mode, which is safe for concurrent writes (though performance may degrade with high contention).

- Role: Implements the core functionality of the class, writing health data to label-specific files in a thread-safe, persistent manner, supporting data logging and analysis.

Technical Points and Design Considerations

1. Persistence and Organization:

- The class organizes data by writing each label (e.g., "ECG", "Alert") to a separate file (`ECG.txt`, `Alert.txt`), simplifying data retrieval and analysis compared to a single file. This structure supports post-simulation processing, such as parsing ECG data or auditing alerts.

- The append-only approach (`StandardOpenOption.APPEND`) ensures data is preserved across simulation runs, suitable for long-term logging.

- The formatted output (`Patient ID: X, Timestamp: Y, Label: Z, Data: W`) is consistent with `ConsoleOutputStrategy`, facilitating debugging and cross-strategy compatibility.

2. Thread Safety:

- The class is designed for concurrent use, leveraging `ConcurrentHashMap` for thread-safe label-to-file mapping. The `computeIfAbsent` method ensures atomic path resolution, preventing race conditions when multiple threads access the same label.

- File writes use `Files.newBufferedWriter` with append mode, which is generally safe for concurrent appends, though high contention (e.g., many threads writing to the same file) could cause performance bottlenecks. Using file locks or a dedicated writer thread could mitigate this.

- The try-with-resources block ensures proper file closure, preventing resource leaks in concurrent environments.

3. Error Handling:

- The class handles I/O errors robustly, catching `IOException` for directory creation and `Exception` for file writes, printing error messages to `System.err`. This ensures the simulator

continues running despite file access issues, though logging to a framework (e.g., SLF4J) would improve traceability.

- Early return on directory creation failure prevents further processing, avoiding redundant errors but potentially missing data for that call. Retrying or falling back to a temporary directory could enhance resilience.

4. Strategy Pattern Integration:

- By implementing `OutputStrategy`, the class adheres to the Strategy Pattern, allowing `HealthDataSimulator` to select file output at runtime (via `--output file:<directory>`). This ensures seamless integration with data generators and other output strategies.

- The generic `output` method parameters support all data types from `PatientDataGenerator` implementations, including alerts from `AlertGenerator`, making it versatile.

5. Integration with Task 7 Design Patterns:

- The `FileOutputStrategy` supports Task 7's alert system by persisting alerts from `AlertGenerator` (e.g., "triggered", "resolved") to files, which can be analyzed offline or integrated with `DataStorage` for processing by `com.alerts.AlertGenerator`. These alerts may be `Alert` objects (from `com.alerts`) created by `AlertFactory` subclasses (e.g., `GenericAlertFactory`) and enhanced by `AlertDecorator` subclasses (e.g., `PriorityAlertDecorator`).

- The class's generic output format aligns with Task 7's patterns, supporting string-based alert data without requiring modifications.

- The Strategy Pattern used by `OutputStrategy` complements Task 7's emphasis on flexible design, allowing alert outputs to be persisted alongside other health data.

6. Potential Improvements:

- Validate `baseDirectory` in the constructor (e.g., checking for empty strings, invalid characters, or write permissions) to throw `IllegalArgumentException` for invalid paths, improving robustness.

- Implement file rotation or size limits to prevent files from growing indefinitely in long-running simulations, enhancing scalability.

- Replace `System.err` error handling with a logging framework (e.g., SLF4J) for structured logging and better integration with monitoring systems.

- Add configurable output formats (e.g., CSV, JSON) via constructor parameters to support diverse analysis needs.

- Optimize concurrent file writes by using a single writer thread or file locks to reduce contention, improving performance in high-throughput scenarios.

- Include metadata in file names (e.g., timestamp, simulation ID) to differentiate outputs from multiple runs.

Interaction with Other Components

- With `OutputStrategy` Interface: The `FileOutputStrategy` implements `OutputStrategy`, providing a concrete implementation of the `output` method. This ensures compatibility with the simulator's strategy-based output system, allowing it to be used interchangeably with other output strategies like `ConsoleOutputStrategy`.

- With `HealthDataSimulator`: The `FileOutputStrategy` is instantiated by `HealthDataSimulator` when file output is selected (via `--output file:<directory>`). It is passed to data generators to handle output during simulation.

- With `PatientDataGenerator` Implementations: Data generators (e.g., `ECGDataGenerator`, `BloodPressureDataGenerator`, `BloodSaturationDataGenerator`, `BloodLevelsDataGenerator`, `AlertGenerator`) invoke the `output` method to write their generated data (e.g., ECG values, blood pressure readings, alerts) to label-specific files.

- With `DataStorage` and `com.alerts.AlertGenerator` (Task 7): The data written to files (e.g., alerts from `AlertGenerator`) can be ingested into `DataStorage` (from `com.data_management`) for processing by `com.alerts.AlertGenerator`, enabling alert generation based on persisted data. This supports Task 7's alert system.

- With `Alert` Class (Task 7): The class persists alert data (e.g., "triggered", "resolved") that may correspond to `Alert` objects (from `com.alerts`) created by `AlertFactory` subclasses and enhanced by `AlertDecorator` subclasses, aligning with Task 7's design patterns.

Summary of Functionality

The `FileOutputStrategy` class is a robust component of the cardiovascular data simulator's output subsystem, implementing the `OutputStrategy` interface to write patient health data to text files. It organizes data by label in separate files, ensures thread-safe access with `ConcurrentHashMap`, and handles directory creation and I/O errors gracefully. Integrated with `HealthDataSimulator` and used by all `PatientDataGenerator` implementations, it supports persistent data storage for analysis and logging. In the context of Task 7, it persists alerts compatible with `AlertFactory` and `AlertDecorator`, contributing to the alert system's functionality. The class's thread safety, error handling, and adherence to the Strategy Pattern make it reliable for simulation scenarios requiring durable storage, though enhancements in validation, logging, and file management could improve its scalability. Overall, `FileOutputStrategy` plays a critical role in enabling persistent data output, supporting both development and analytical use cases in the simulator.