Overview of the `WebSocketOutputStrategy` Class

Package: `com.cardio_generator.outputs`

Purpose:

The `WebSocketOutputStrategy` class is responsible for sending patient health data to multiple WebSocket clients in the cardiovascular data simulator. It implements the `OutputStrategy` interface, establishing a WebSocket server on a specified port and broadcasting data in a comma-separated format to all connected clients. The class supports real-time, bidirectional data streaming, enabling remote monitoring or integration with web-based applications, such as clinical dashboards or real-time visualization tools.

Role in the Project:

The `WebSocketOutputStrategy` is a key component of the simulator's output subsystem, alongside `ConsoleOutputStrategy`, `FileOutputStrategy`, and `TcpOutputStrategy`. It is used by data generators (e.g., `ECGDataGenerator`, `BloodPressureDataGenerator`) via the `HealthDataSimulator` when WebSocket output is selected (e.g., via the `--output websocket:<port>` command-line argument). In the context of Task 7, it broadcasts alerts generated by `AlertGenerator`, which may be structured as `Alert` objects (from `com.alerts`) and processed by `AlertFactory` or `AlertDecorator`, supporting real-time alert monitoring across multiple clients. The class is ideal for scenarios requiring scalable, real-time data delivery to web or networked clients, enhancing the simulator's applicability in modern healthcare monitoring systems.

Technical Characteristics:

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

- It uses the `org.java_websocket` library to create a WebSocket server, supporting multiple simultaneous client connections.

- Data is sent in a comma-separated format (`patientId,timestamp,label,data`), optimized for parsing by clients.

- The class is designed for concurrent use, leveraging the WebSocket library's thread-safe connection management, though it lacks explicit error handling for data transmission failures.

Internal Components and Their Purposes

The `WebSocketOutputStrategy` class consists of a field, a constructor, a method, and an inner class. Each component is described below, including its purpose, technical details, and role in the class's functionality.

1. Field:

- `private WebSocketServer server`

- Purpose: Represents the WebSocket server that listens for client connections and manages data broadcasting.

- Technical Details:

- Type: `WebSocketServer`, a base class from the `org.java_websocket` library, extended by the inner `SimpleWebSocketServer` class.

- Private access ensures encapsulation, initialized in the constructor with a specific port.

- Used to manage client connections and send data via the `output` method.

- Role: Serves as the core component for WebSocket communication, handling client connections and data transmission to all connected clients.

2. Constructor:

- `public WebSocketOutputStrategy(int port)`

- Purpose: Initializes a `WebSocketOutputStrategy` instance by creating and starting a WebSocket server on the specified port.

- Technical Details:

- Parameter: `port` (`int`), the port number (1–65535) on which the server will listen.

- Logic:

- Instantiates a `SimpleWebSocketServer` with an `InetSocketAddress` for the specified port.

- Prints a confirmation message to `System.out`: `WebSocket server created on port: <port>, listening for connections...`.

- Calls `server.start()` to begin listening for client connections, running the server in its own thread (managed by the WebSocket library).

- Public access allows instantiation by `HealthDataSimulator` when WebSocket output is configured.

- Does not validate `port` explicitly, relying on the WebSocket library to handle invalid ports (e.g., throwing `IllegalArgumentException` or `IOException`).

- Does not catch exceptions from `server.start()`, assuming successful startup (though `SimpleWebSocketServer`'s `onError` handles runtime errors).

- Role: Sets up and starts the WebSocket server, preparing it to accept multiple client connections and broadcast health data.

3. Method:

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

- Purpose: Broadcasts patient health data to all connected WebSocket clients in a comma-separated format (`patientId,timestamp,label,data`).

- Technical Details:

- Parameters:

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

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

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

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

- Return Type: `void`, as the method's effect is sending data to clients.

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

- Logic:

  - Formats the data using `String.format` into `<patientId>,<timestamp>,<label>,<data>`.

  - Iterates over all connected clients (`server.getConnections()`) and sends the message to each using `conn.send(message)`.

  - Silently ignores cases where no clients are connected (empty `getConnections()`), avoiding errors.

- Thread Safety: The `org.java_websocket` library ensures thread-safe connection management, and `conn.send` is safe for concurrent calls, though high-frequency broadcasting could strain server resources.

- Role: Implements the core functionality of the class, broadcasting health data to all connected WebSocket clients in a parseable format, supporting real-time monitoring.

4. Inner Class:

- `private static class SimpleWebSocketServer extends WebSocketServer`

- Purpose: Extends `WebSocketServer` to provide custom behavior for WebSocket events (e.g., client connections, disconnections, errors), logging connection status and errors to `System.out` or `System.err`.

- Technical Details:

  - Type: Static inner class extending `WebSocketServer` from `org.java_websocket`.

  - Private and static, restricting instantiation to within `WebSocketOutputStrategy` and allowing independent use if needed.

  - Constructor: Takes an `InetSocketAddress` to specify the server's address and port.

  - Overridden Methods:

    - `onOpen(WebSocket conn, ClientHandshake handshake)`: Logs new client connections with their remote address.

    - `onClose(WebSocket conn, int code, String reason, boolean remote)`: Logs client disconnections with their remote address.

- `onMessage(WebSocket conn, String message)`: Empty implementation, as the server does not process incoming messages in this context.

- `onError(WebSocket conn, Exception ex)`: Prints stack traces for errors (e.g., connection failures) to `System.err`.

- `onStart()`: Logs a confirmation message when the server starts successfully.

- Thread Safety: Inherits thread-safe connection handling from `WebSocketServer`, suitable for concurrent client management.

- Role: Customizes the WebSocket server's behavior, providing logging for connection events and errors, enhancing debugging and monitoring capabilities.

Technical Points and Design Considerations

1. Real-Time Data Broadcasting:

- The class broadcasts data in a comma-separated format (`patientId,timestamp,label,data`), optimized for easy parsing by WebSocket clients (e.g., web browsers, clinical dashboards). This format is consistent with `TcpOutputStrategy`, facilitating client-side compatibility across output strategies.

- The use of WebSocket enables bidirectional, low-latency communication, ideal for real-time applications, unlike `FileOutputStrategy`'s persistence focus or `ConsoleOutputStrategy`'s debugging purpose.

2. Multi-Client Support:

- Unlike `TcpOutputStrategy`'s single-client limitation, `WebSocketOutputStrategy` supports multiple simultaneous clients via `server.getConnections()`, broadcasting data to all connected clients. This scalability suits scenarios with multiple monitoring stations or users.

- The `org.java_websocket` library manages client connections efficiently, handling connection lifecycles (open, close, error) in separate threads, ensuring non-blocking operation.

3. Thread Safety and Concurrency:

- The class leverages the `org.java_websocket` library's thread-safe connection management, with `server.getConnections()` and `conn.send` designed for concurrent access. This aligns with `HealthDataSimulator`'s multi-threaded execution, where multiple threads may call `output` simultaneously.

- The `output` method's iteration over connections is safe, but high-frequency broadcasting to many clients could strain server resources. Rate limiting or batching messages could optimize performance.

4. Error Handling:

- The constructor does not catch exceptions from `server.start()`, assuming successful startup, though `SimpleWebSocketServer`'s `onError` method handles runtime errors by printing stack traces to `System.err`. Structured logging (e.g., SLF4J) would improve error traceability.

- The `output` method silently ignores cases with no connected clients, avoiding errors but potentially losing data. Logging ignored data or queuing it for later transmission could enhance reliability.

- No explicit validation for `port` or input parameters (`patientId`, `label`, `data`) is performed, relying on callers and the WebSocket library to handle invalid inputs.

5. Strategy Pattern Integration:

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

- The `output` method's generic parameters support all data types from `PatientDataGenerator` implementations, including alerts, making it versatile for broadcasting health data and alerts.

6. Integration with Task 7 Design Patterns:

- The `WebSocketOutputStrategy` supports Task 7's alert system by broadcasting alerts from `AlertGenerator` (e.g., "triggered", "resolved") to WebSocket clients, enabling real-time alert visualization. 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 comma-separated format is compatible with Task 7's alert processing, allowing clients to parse alert data for display or analysis.

- The Strategy Pattern used by `OutputStrategy` aligns with Task 7's emphasis on flexible design, enabling alert broadcasting alongside other health data.

7. Potential Improvements:

- Validate `port` in the constructor (e.g., checking for 1–65535 range) to throw `IllegalArgumentException` for invalid values, improving robustness.

- Add error handling in `output` for transmission failures (e.g., catching `RuntimeException` from `conn.send`), logging failures and attempting reconnection for affected clients.

- Implement message queuing to handle temporary client disconnections, ensuring no data loss.

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

- Introduce configurable data formats (e.g., JSON, XML) via constructor parameters to support diverse client needs.

- Add support for incoming messages in `onMessage` to enable client-server interaction (e.g., requesting specific data types), enhancing interactivity.

- Implement server shutdown logic (e.g., a `close` method to stop `server` and release resources) to ensure proper cleanup when the simulator terminates.

Interaction with Other Components

- With `OutputStrategy` Interface: The `WebSocketOutputStrategy` 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`, `FileOutputStrategy`, and `TcpOutputStrategy`.

- With `HealthDataSimulator`: The `WebSocketOutputStrategy` is instantiated by `HealthDataSimulator` when WebSocket output is selected (via `--output websocket:<port>`). 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 broadcast their generated data (e.g., ECG values, alerts) to all connected WebSocket clients.

- With `DataStorage` and `com.alerts.AlertGenerator` (Task 7): While `WebSocketOutputStrategy` broadcasts data in real-time, the data (e.g., alerts from `AlertGenerator`) can be ingested into `DataStorage` (from `com.data_management`) by a client or secondary process for processing by `com.alerts.AlertGenerator`, supporting Task 7's alert system.

- With `Alert` Class (Task 7): The class broadcasts 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 `WebSocketOutputStrategy` class is a critical component of the cardiovascular data simulator's output subsystem, implementing the `OutputStrategy` interface to broadcast patient health data to multiple WebSocket clients. It creates a WebSocket server, supports multiple simultaneous connections, and sends data in a comma-separated format, enabling real-time monitoring. Integrated with `HealthDataSimulator` and used by all `PatientDataGenerator` implementations, it facilitates scalable, web-based data delivery for clinical or analytical applications. In the context of Task 7, it broadcasts alerts compatible with `AlertFactory` and `AlertDecorator`, supporting real-time alert visualization. The class's multi-client support, thread-safe connection management, and adherence to the Strategy Pattern make it effective for networked scenarios, though enhancements in error handling, logging, and resource cleanup could improve its reliability and scalability. Overall, `WebSocketOutputStrategy` plays a

pivotal role in enabling real-time, scalable data streaming, enhancing the simulator's applicability in modern healthcare monitoring.