Title: Report Document

Author: Nam Cap

Student ID: a1907345

1. **Functional Reactive Programming (FRP) Usage:**

- I use of the Sodium library for FRP seems appropriate. I am creating streams and cells, and using them to react to events. This is a core part of FRP, and your implementation aligns well with this paradigm.
- Ensure that all data processing and transformations required for the GPS data (like filtering and distance calculations) are done reactively using the Sodium library.

2. **Implement the refactored GUI setup:** We'll ensure that the refactoring of createAndShowGUI and related methods is integrated into your GpsGUI class.

3. **Implement filter logic:** We'll write the logic for filtering the stream of GPS events based on user input.

```
private static void
setupFilterButtonListener(STextField latitudeField,
STextField longitudeField,
            SButton applyButton, JLabel statusLabel) {
        // Apply filter logic on button click
        applyButton.sClicked.listen(ignored -> {
            String latStr =
latitudeField.text.sample().trim();
            String lonStr =
longitudeField.text.sample().trim();
            try {
                double lat =
Double.parseDouble(latStr);
                double lon =
Double.parseDouble(lonStr);

                // Validate latitude and longitude
values
```

```
                if (lat < -90 || lat > 90 || lon < -
180 || lon > 180) {
                    JOptionPane.showMessageDialog(fram
e,
                            "Latitude must be between
-90 and 90 and longitude between -180 and 180.");
                    statusLabel.setText("Current
filter: Invalid");
                    return; // Exit the method if the
input is invalid
                }

                // If the input is valid, update the
filter status label and apply the filter
                filterStatusLabel.setText("Current
filter: Lat " + lat + ", Lon " + lon);

                // Filter the stream for events within
the specified latitude and longitude
                Stream<GpsEvent> allEventsStream =
combineAllTrackerStreams();
                Stream<GpsEvent> filteredStream =
allEventsStream
                        .filter(event ->
Math.abs(event.getLatitude() - lat) <
LATITUDE_THRESHOLD &&
                                Math.abs(event.getLong
itude() - lon) < LONGITUDE_THRESHOLD);

                // Update the combined data display
with the filtered data
                filteredStream
```

```
                        .map(event -> "Tracker " +
event.getTrackerId() + ": Lat " + event.getLatitude()
+ ", Lon "
                                        +
event.getLongitude())
                        .hold("No data")
                        .listen(filteredData ->
combinedDataDisplay.setText(filteredData));
            } catch (NumberFormatException e) {
                // If parsing the double fails, show
an error message
                JOptionPane.showMessageDialog(frame,
"Invalid input for latitude or longitude.");
                statusLabel.setText("Current filter:
Invalid");
            }
        });
    }
```

4. **Implement Distance Update Logic**
   The distance update logic is likely to be complex as it involves geospatial calculations. You'll need to implement a method to calculate the distance between two GPS coordinates, possibly including altitude if required. The calculateDistance method should then be used to update the distance displayed for each tracker.

```
public static double calculateDistance(GpsEvent
startEvent, GpsEvent endEvent) {
        final int R = 6371; // Radius of the Earth in
kilometers

        double latDistance =
Math.toRadians(endEvent.getLatitude() -
startEvent.getLatitude());
        double lonDistance =
Math.toRadians(endEvent.getLongitude() -
startEvent.getLongitude());
```

```
        double a = Math.sin(latDistance / 2) *
Math.sin(latDistance / 2) +
                Math.cos(Math.toRadians(startEvent.get
Latitude())) *
Math.cos(Math.toRadians(endEvent.getLatitude()))
                        *
                        Math.sin(lonDistance / 2) *
Math.sin(lonDistance / 2);
        double c = 2 * Math.atan2(Math.sqrt(a),
Math.sqrt(1 - a));

        double distance = R * c; // convert to meters
        distance *= 1000; // convert to meters

        return Math.round(distance);
    }
```

5. **Write test cases:** We'll write additional test cases in Gui_Test.java to ensure the GUI behaves as expected.

I wrote in "Test Case.pdf", please review it for more information.

6. **The setupPeriodicTasks method**

The setupPeriodicTasks method schedules a periodic task that will update each tracker's distance display every five minutes. This is an important piece of functionality that ensures the GUI reflects the most recent state of the system.

```
private static void setupPeriodicTasks() {
        ScheduledExecutorService executorService =
Executors.newScheduledThreadPool(1);
        executorService.scheduleAtFixedRate(() -> {
            // For each tracker, update its distance
display
            trackerDistances.forEach((trackerId,
distance) -> {
```

```
                updateTrackerDistanceDisplay(trackerId
, distance);
            });
        }, 0, 5, TimeUnit.MINUTES); // Schedules the
task to run every 5 minutes
    }
```

7. **Logic for Latitude and Longitude**

- Latitudes range from -90 to 90.
- Longitudes range from -180 to 180.

```
private static void simulateTrackerData(String
trackerId, StreamSink<GpsEvent> sink) {
        Random rand = new Random();
        Timer timer = new Timer(true); // Create a
daemon thread that will not prevent the application
from exiting
        timer.scheduleAtFixedRate(new TimerTask() {
            public void run() {
                // Generate random latitude and
longitude with 8 decimal places
                double latitude = -90 + 180 *
rand.nextDouble(); // Range: -90 to 90
                double longitude = -180 + 360 *
rand.nextDouble(); // Range: -180 to 180
                double altitude = 1000 *
rand.nextDouble(); // Random altitude up to 1000
meters

                // Round to 8 decimal places for
latitude and longitude
                latitude = Math.round(latitude * 1e8)
/ 1e8;
```

```
                longitude = Math.round(longitude *
1e8) / 1e8;

                // Create a new GpsEvent with
randomized data
                GpsEvent event = new
GpsEvent(trackerId, latitude, longitude, altitude);

                // Print the generated event for
debugging
                System.out.println("Generated event
for " + trackerId + ": " + event);

                // Send the new event through the
StreamSink
                sink.send(event);
            }
        }, 0, 1000); // Schedule to run every second
    }
```

With these ideas (more will be say in "Test Case.pdf" document) and code, I already clear all the requirements from the question for simulated live GPS data from a pool of devices with several applications that make use of that data in different ways/formats, displaying the results of processing that data in a Java GUI using the Sodium and Swidget libraries.