Event Driven Computing

Assignment 3 – Testing Document

Spencer Charles Edwards - a1765159

Instructions to run my code:

```
// create directory for class binaries in root repository
EDC-ass3$ mkdir bin

// compile from src directory
EDC-ass3/src$ javac -d ../bin -cp :../lib/*:../bin: myGUI.java

// execute from src directory
EDC-ass3/src$ java -cp :../lib/*:../bin: myGUI
```

Instructions to run my tests:

```
// compile all tests from src directory (after creating bin directory)
EDC-ass3/src$ javac -d ../bin -cp :../lib/*:../bin: *_Test.java

// run specific test from src directory
EDC-ass3/src$ java -cp :../lib/*:../bin: org.junit.runner.JUnitCore simplifiedGps_Test
EDC-ass3/src$ java -cp :../lib/*:../bin: org.junit.runner.JUnitCore allEvents_Test
EDC-ass3/src$ java -cp :../lib/*:../bin: org.junit.runner.JUnitCore ControlPanel_Test
EDC-ass3/src$ java -cp :../lib/*:../bin: org.junit.runner.JUnitCore filteredEvents_Test
EDC-ass3/src$ java -cp :../lib/*:../bin: org.junit.runner.JUnitCore distTravelled_Test
```

Testing My Code

simplifiedGps_Test

The requirements for the first part of the GUI are:

1. Ten tracker simplified displays. Simplified tracking data has its altitude data removed and this should be carried out with Sodium FRP operations. From this stream, display, in separate cells:
   - Tracker number
   - latitude
   - longitude

Implementation

My final implementation to meet these requirements is the getSimplifiedGpsCells() function that takes a Stream of GpsEvents as a parameter and returns an array of Cells that hold the required fields of the Stream's events. To strip the altitude from the GpsEvent, I created a new class called SimpleGpsEvent which does not have the altitude, and a function stripAltitude() that maps the GpsEvent Stream to a SimpleGpsEvent Stream. The function uses a Cell to hold the mapped Stream's events and returns 3 other Cells that mapped the current SimpleGpsEvent to each of the required fields.

Testing Goals

To show that my code does what I say it does, I think it is sufficient to test the following two parts of my implementation:

1. The altitude data is removed using Sodium FRP operations
2. The function's returned cells hold the correct data

Tests

stripAltitide_test():

Although, we can check that the SimpleGpsEvent class does not have an altitude field, I decided to be thorough and check that the stripAltitude() function correctly carries the desired fields over. To do this, the test uses a StreamSink to pass to the function. By sending an event to the StreamSink, the test verifies that the SimpleGpsEvent Stream's values are as expected, and that none of them are the altitude.

getSimplifiedGpsCells_test():

This test also uses a StreamSink as a parameter for the getSimplifiedGpsCells() function, and sends two distinct GpsEvents down it. The test then verifies that the returned Cells updated to hold the event's fields correctly, after each event is sent.

allEvents_Test

The requirements for the second part of the GUI are:

> 2. A display field shows each event as it is passed to the GUI, at the time it occurs.
>   - This is presented as a comma-delimited string of 4 items
>   - Only a single entry is presented at a time, and cleared after 3 seconds if not overwritten.

Implementation

My final implementation to meet these requirements is the getAllEventsCell() function that takes as parameters: an array of GpsEvent Streams, and a SecondsTimerSystem. The SecondsTimerSystem is a Sodium FRP timer system implementation that contains a clock that holds the number of seconds since program start. The function returns a Cell that holds the most recent GpsEvent presented as a comma-delimited string of all its fields. The Cell will also be cleared once approximately 3 seconds pass without an event occurrence, though this does not really occur when running the program.

The Streams of the array are merged into one using orElse(), a variant of Sodium's merge() method. It also ensures that in the occurrence of simultaneous events, only one is fired. Thus, it is easy to return a Cell that simply holds the latest event from the merged stream in the form of a string. To clear the cell after approximately 3 seconds, I used a function periodic() from the FRP book that creates a Stream that fires an event repeatedly in set intervals. I chose a period of 0.1 seconds, which is why I have been saying approximately 3 seconds. In addition to the timer, I use a Cell that stores the time of the last GpsEvent occurrence. There is a second Stream that uses the periodic Stream to check the latest event time, filtering out the periodic Stream's events if it has not been 3 seconds, and mapping its events to an empty string in the case that it has been 3 seconds. Finally, the GpsEvent string Cell merges the merged GpsEvent Stream with the Stream that sends an empty string.

Testing Goals

I think it is sufficient to test the following:

1. The streams are merged correctly
2. The GpsEvent string Cell is cleared after approximately 3 seconds

Tests

mergeStreams_test():

This test creates an array of StreamSinks and passes it as a parameter for the getAllEventsCell() function. The test then sends a GpsEvent to each StreamSink one after the other. Between sending the events, the test verifies that the Cell updates to the correct GpsEvent String.

clearStream_test():

This test also creates an array of StreamSinks for the getAllEventsCell() function. After sending an event, the test waits 2.5 seconds and checks that the Cell has not been cleared. After 0.6 more seconds, it verifies that it has been cleared. This test and future tests that involve time, uses Thread.sleep() to wait for a specified time, this led to some issues with the code being updated, but it has been handled using some suggestions from the book. The tests pass on my system, but it might be possible that it won't in a different environment.

ControlPanel_Test

The third and fourth parts of the GUI require a control panel consisting of a latitude input field, a longitude input field, and a button to set the restriction. It also displays the current restrictions.

Implementation

My final implementation of this GUI component is a ControlPanel class that has its current restrictions held in Cells and made public. Its text fields are also public, so that it can be changed during testing. By default, the Cells hold the widest range possible, i.e., -90 to 90 for latitude and -180 to 180 for longitude. To help simulate the button-click in tests, the constructor takes a Unit Stream as a parameter, and uses it throughout the code in the case that the Stream is not null.

The class uses a modified Rule class from the book to encapsulate business rules. The business rule the class defines, ensures that the button cannot be clicked if the current input is invalid. The rule uses the checkCoord() function to verify an input pair is a valid latitude or longitude pair, depending on the type parameter. When the button is clicked, the Cells use the snapshot() primitive to capture the value of each input field at the time of the click. In addition to this, a button-click clears the input fields using a simple map() primitive.

Testing Goals

I think it is sufficient to test the following:

1. Valid and invalid input handling
2. Cells are updated correctly on the click of the button
3. Input fields are cleared on the click of the button (Optional, it's not part of the spec)

Tests

validLatitudeInput_test() and *invalidLatitudeInput_test()*

There is not much to test for valid inputs, so the result of checkCoord() is verified for inputs that check decimal usage, sign usage, and the latitude boundaries. For invalid inputs, the result of checkCoord() is verified for inputs that check out of boundary cases, cases when the minimal value is larger than the maximal, and non-numeric character usage.

validLongitudeInput_test() and *invalidLongitudeInput_test()*

These tests do the same thing as above, but for the longitude.

applyRestrictions_test()

To simulate the button-click, this test creates a StreamSink as a parameter for the ControlPanel constructor. The test sets the values of the control panel's input text fields and verifies that the Cells containing the restrictions are still the default values, and that the input fields have been set. It then sends a Unit to the StreamSink, simulating a button click. The Cells are then verified to be the values that the text fields were set to, and the text fields are verified to have been cleared.

filteredEvents_Test

The requirements for the third part of the GUI are:

> 3. A single (1) display that combines all input streams, and only outputs GPS events in the defined range of Latitude and Longitude.
>   - The defined range is controlled by a control panel consisting of a latitude input field, a longitude input field, and a button to set the restriction.
>   - The latitude and longitude restriction will probably require the use of the **snapshot** primitive.
>   - Labels show the current setting of the restriction.
>   - The GUI element displays the data in an identical form to the display field in (2), as long as the data is in the range.

Implementation

My final implementation to meet these requirements is the getFilteredEventsCell() function which takes in the following parameters: SecondsTimerSystem, ControlPanel, and an array of GpsEvent Streams. Since this display is nearly the same as the second display, much of the code from getAllEventsCell() was reused. As a result, much of the explanation of my implementation is the same as well. The only difference is the addition of the ControlPanel, whose restriction Cells should be used to restrict which GpsEvents the returned Cell will hold.

To do this, a second Stream is created using a series of filter() primitives on the merged GpsEvent Stream. The filters check that the event's latitude and longitude are within the range of the ControlPanel's Cells. This stream is then used in the construction of the GpsEvent String Cell. The implementation of the timer that clears the Cell after 3 seconds of no event occurrences remains the same. When running the code, the restrictions can be set such that we finally get to see the display be cleared!

Testing Goals

Since most of the functionality of this display was code that has already been tested, I think it is sufficient to test only the following:

1. Events are filtered out according to the current specified restrictions

Tests

getFilteredEventsCell_test():

This test creates the StreamSinks and initializes the GpsEvents that will be sent down the StreamSinks,, creates the ControlPanel it will update, and creates the Cell we want to test. The Cell is verified to contain the string form of all 6 GpsEvents that are sent to the StreamSinks, one after another.

The test then sets and updates the ControlPanel's restrictions. One after another, the same 6 events are sent down the StreamSinks. This time, only 2 of the events are within the ControlPanel's new range. This is verified by checking the Cell not only doesn't update to out of range GpsEvents, but still holds the string value of the last in range GpsEvent.

distTravelled_Test

The requirements for the fourth part of the GUI are:

> 4. A distance field for each tracker that outputs the total distance travelled over the last 5 minutes for each tracker.
>   - Your GUI should display a label for each tracker, with the current distance travelled next to it.
>   - This only includes events that are within the currently set Lat/Long range.
>   - The value should be a distance in meters, rounded up to the nearest integer meter.
>   - **Important: The tracker altitude data is in feet. You should use `map` to transform this data.**
>   - Distance calculations may be calculated without considering the curvature of the Earth but must take the altitude into account.
>   (See Latitude and Longitude section below)
>   - **Hint:** You might want to look at the FRP primitive `snapshot`.
>   - **Hint:** A sliding window of 5 minutes, however you implement it, will make this calculation easier. You only calculate distances between known positions, i.e. two distinct events

Implementation

My final implementation to meet these requirements is the getDistTravelledCell() function which takes in the following parameters: SecondsTimerSystem, ControlPanel, and a Stream of GpsEvents. The function returns a Cell that holds the distance travelled within the last 5 minutes between GpsEvents that are in the control panel's current restrictions. To implement a sliding window, I use the function getSlidingWindow() which takes in the same parameters above, in addition to an integer that specifies the size of the window in seconds. The function returns a sliding window in the form of a Stream of ArrayList<GpsEvent> events.

The getSlidingWindow() function creates a Cell that holds an ArrayList of GpsEvents. The Cell accumulates events from the Stream using the Sodium accum() function which allows us to update the state of a Cell upon each Stream event, which I use to add the event to the list. Since the window size is not configurable by the user, it is not efficient to keep events that are not within the sliding window anymore. To account for this, the cell also maps its current list to a new list that does not contain events outside of the sliding window. It does not remove events out of the control panel's restrictions because the restrictions can be changed. Events may move out of the sliding window, but it will not be removed from the Cell until the Cell receives a new event. To account for this, a periodic Stream is created to fire an event every second. Finally, the Stream to be returned is created. It uses the snapshot() primitive on the periodic Stream to capture the list of the Cell, before it outputs a list of events that are both within the sliding window, and the control panel's restrictions.

In getDistTravelledCell, a Cell uses the sliding window's list to map to a distance, using the list as a parameter for another function calcDistance(). The calcDistance() function returns 0 if the list's size is less than 2 because at least 2 distinct positions are needed to get a distance. The function then calculates the distance travelled between the events in the list using calcEuclidean(), before returning the distance as a rounded up integer.

The calcEuclidean() function takes in the latitude, longitude, and altitude of two points as parameters. It converts these coordinates into Earth-centered, Earth-fixed (ECEF) coordinates,

which is in the form X,Y,Z. It then applies the standard Euclidean formula to return the distance in meters between these points.

The convertToECEF() function takes latitude, longitude, and altitude as parameters. The altitude is given in feet, so this function converts it to meters. It then uses formulae to convert the coordinates to ECEF coordinates, returning it as an array of 3 Doubles.

Testing Goals

Since the required functionality was abstracted into multiple functions, I think it's more than sufficient to test the following:

1. The sliding window does not contain events older than the specified window size
2. The sliding window does not contain events outside of the control panel's restrictions
3. The conversion to ECEF coordinates is relatively accurate
4. The Euclidean distance between two GpsEvent positions are relatively accurate
5. The distance of a list of less than two events, or events at the same position is 0
6. The distance of a list of events matches the expected output
7. The Cell contains the expected distance while changing the control panel's restrictions

Tests

slidingWindow_test1():

This test verifies that the sliding window does not contain an event that has moved out of the specified time window. It does this by sending two events to a StreamSink, with an interval between each other. The interval is created using Thread.sleep() before sending the second event. After the second event is sent, the test verifies that the sliding window contains both events. The test then sleeps for a time until there should only be 1 event in the sliding window and verifies that this is the case.

slidingWindow_test2():

This test verifies that the sliding window does not contain events outside of the control panel's restrictions. 6 events are sent down a StreamSink before verifying that the sliding window contains all 6 events, since the control panel's default is the widest range possible. The restrictions are then changed multiple times in the test, verifying that only the expected events remain in the sliding window after each change. The test then changes the restrictions back to its default value and verifies that all 6 events are in the sliding window again.

convertToECEF_test():

Since I cannot use any other external libraries in this assignment, I opted to compare my results with an online calculator's. The test calculates the ECEF of a point and checks that the result is equal to the return value of calculator, with a margin of error of 1 meter. It does the same for another point with negative values, just to check that the function is working appropriately.

calcEuclidean_test():

This test uses another online calculator to compare my results with. The distance between two points was verified with a margin of error of 1 meter. The test also verifies the result of another pair of points, with the only difference in the points being a change in altitude of 1 meter. The distance between them should therefore only be 1 meter, this was verified with a margin of error of 0.01 meters.

calcDistance_test1():

This test verifies that the returned distance is 0 for event lists of less than 2. It also verifies that the distance between two different-positioned points is non-zero.

calcDistance_test2():

This test verifies that the distance between two points of the same positions is 0. It then verifies that the calcDistance() function is correctly summing up the distance calculated between multiple events.

getSimplifiedGpsCells_test():

This test may seem a little redundant since the sliding window and functions that do the actual calculations of distance have already been tested, but I think it's good to test that they are all working well together. The test checks this by first calculating a few distances between events, which will be the different expected results as we change the control panel's restrictions. The distance between all the events is first verified before changing the restrictions such that the sliding window contains a few different combinations of events, verifying that the Cell holds the expected distance each time. Finally, the restrictions are reset to default before verifying the distance in the Cell is the same as it was at the start of the test.