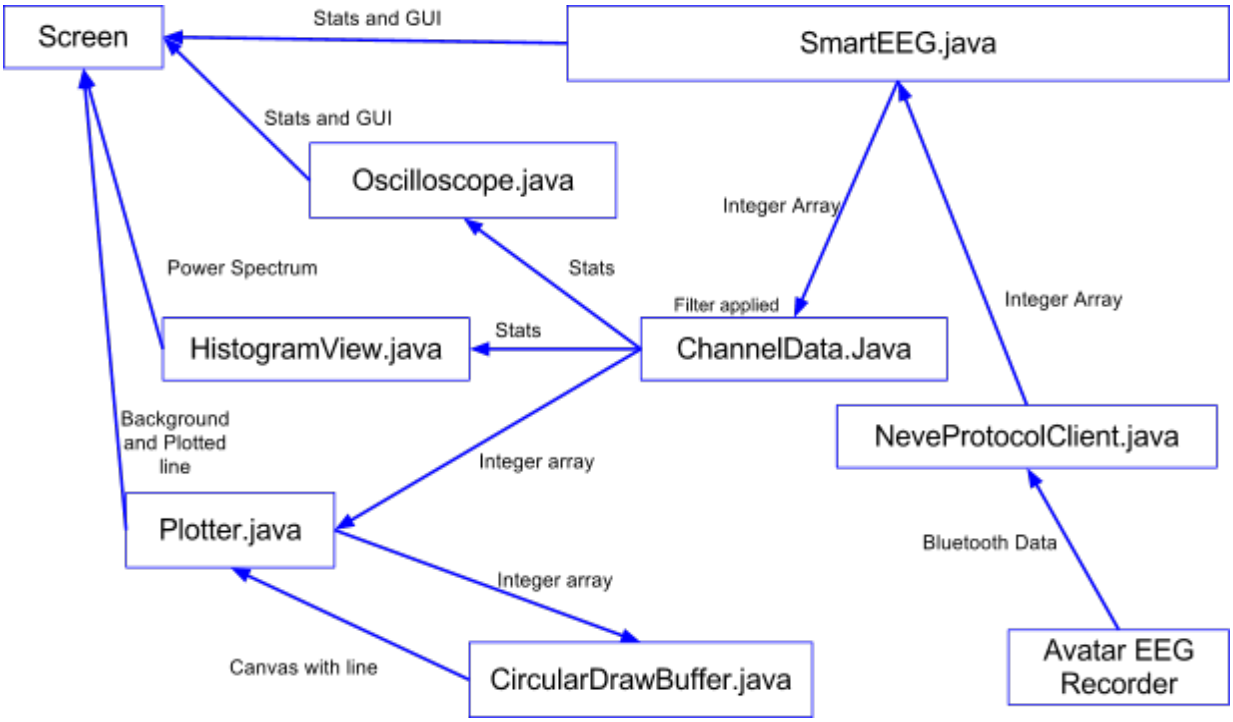
Program overview:

## Data Flow




**SmartEEG.java** controls everything. It creates an instance of NeveProtocolClient.java to handle connecting and receiving data, a hashmap of 8 ChannelData.java objects for holding data, and a list of up to 8 oscilloscopes for printing data. After initial startup, SmartEEG starts NeveProtocolClient.java (from within setupOscilloscope()).

- **NeveProtocolClient.java** spawns threads to handle connecting to and reading data from an AvatarEEG recorder, or to generate test data if the testMode variable it receives from SmartEEG is set to true. Once it has read or generated data for a channel, NeveProtocolClient.java sends MESSAGE_READ_DATA to SmartEEG, along with an array of integer values. After sending data for all channels, Neve sends MESSAGE_GFX_UPDATE to SmartEEG.

Upon receipt of MESSAGE_READ_DATA, **SmartEEG** inserts the received data into the hashmap of ChannelData.java objects by calling ChannelData's newData() method. Each ChannelData within the hashmap holds the data for one channel, and its position within the hashmap is the only indicator of which channel the data belong to.

- **ChannelData.java** implements a circular buffer. If the filter is enabled, it applies it as it inserts new data into the buffer. It also handles calculating statistics (vpp, offset, etc.)
    - **ChannelDataOverflow.java** is assumably used by ChannelData, although it may now be obsolete

After inserting data into the ChannelData hashmap, **SmartEEG** checks for an oscilloscope that is displaying data on the channel in question, and passes the channeldata hashmap in to this oscilloscope through the drawAll() method (this name is a holdover from a previous version of the app, and should probably be changed).

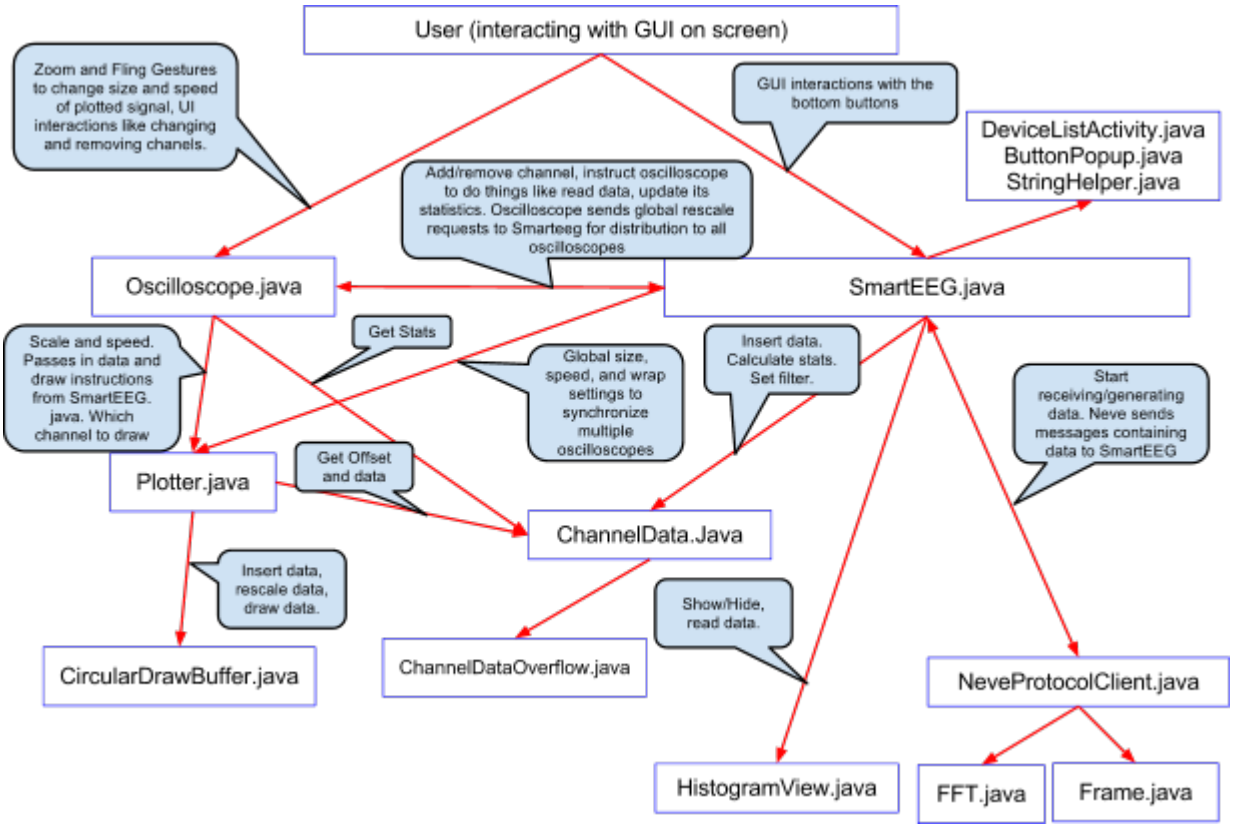
- **Oscilloscope.java** handles all aspects of an oscilloscope. It contains the controls that appear on each oscilloscope, such as the channel select spinner, exit button, and stats display, as well as the detectors for scale and speed changing gestures. It also contains a Plotter.java object, which draws the plotted line and the background. When Oscilloscope receives the channeldata object, it keeps a reference to it and then passes it along with the current channel to Plotter.java's dataUpdate() method.
    - **Plotter.java** handles drawing of the plotted line and background. Upon receiving data in its dataUpdate() method, plotter applies its scale factor (for adjusting the amplitude of the signal), its offset (which is the median value, determined in the calculateStats() method of ChannelData), and its midline (which adjusts zero values of the signal to the middle of the plotter), and then writes the data into a CircularDrawBuffer.java object.
        - **CircularDrawBuffer.java** implements a circular buffer, in the format required by android's DrawLines() method. DrawLines expects an array of start and end points for its lines, and because the plotted line is unbroken from left to right, in our case that format is: {$x_1$, $y_1$, $x_2$, $y_2$, $x_2$, $y_2$, $x_3$, $y_3$ ...}, meaning that each point, except for the first and last, must be stored twice. Because the line is drawn from left to right across the screen, the x values simply correspond to pixel values from 0 to the width of the screen. Plotter therefore passes the screen width in to CircularDrawBuffer, which sets its width accordingly.[1]
        CircularDrawBuffer handles the rather complicated task of inserting new points into this buffer, and then draws said buffer to a canvas when given one by Plotter.

Finally, after every 1000 data points **SmartEEG** sends to the uppermost oscilloscopeès ChannelData, SmartEEG instructs ChannelData to calculate its statistics, including a power spectrum (using a sample size defined in FFT.java). SmartEEG then instructs each Oscilloscope to update its displayed statistics, and passes this power spectrum to HistogramView.java

- **HistogramView.java** draws a scale of Hz on the vertical axis and 0.5 to 0 on the horizontal, and then draws the power spectrum as a histogram, with lines coloured to correspond to different brain wave frequencies.

## Control Flow

[1] To accommodate duplicate x and y values, the width is calculated as <screen width> * 4 - 4. four times the screen width minus two points at the start and end of the buffer.




# Notes

## Eclipse tips:

- to change keybindings in eclipse, go to Window -> preferences -> general -> keys
- to fix the “Unable to resolve target, ‘android-x’” error, right click on project and go properties -> android, then pick an available build.
    - Also fix the target in the androidmanifest.xml file
    - also in project.properties
- you may also need to ensure that your java compiler is 1.6 instead of 1.5. Check this in project properties -> java compiler
- “weight” in linear layouts controls how much of the screen is shared between two views. heavier weights mean less screen, apparently. LINT throws a warning about nesting weights, which is probably why the other GUI xml file uses relative layouts. However, nesting a horizontal inside a vertical is done in the linear layout tutorial, and seems ok.
- the “R cannot be resolved” error is really annoying. I’ve been able to fix it by deleting the gen folder and restarting eclipse a bunch

## Tidbits about the app itself:

- the vertical (“x”) axis of the fft is frequencies, ranging from 0 to the number of samples (128). The horizontal (“y”) is microvolts