EECS293 Project 11 – Filter Design Document - Shaun Howard (smh150)

Given the assignment document, I have developed the following design for the filter system.

Filter → MaxFilter, MinFilter, MaxFilterN, MinFilterN, FilterCascade
FilterN → MaxFilterN, MinFilterN, AveragingFilterN
IdentityFilter
ScalarFilter → AveragingFilter, AveragingFilterN, ScalarLinearFilter
ScalarLinearFilter → FIRFilter
FIRFilter → GainFilter, BinomialFilter

A Filter is a generic interface with two type parameters, A and B.
A must extend a comparable class in order to be compared during filtering.
The input is typed must be of type A and the output will be of type B.
The filter can be reset with a given value of type A.
The following are the method signatures of the Filter interface:

```
    /**
     * Filters the input value of type A based on the filter type.
     *
     * @param value - the value to filter
     * @return the filtered value of type B
     */
    public B filter(A value);

    /**
     * Resets the filter with the input value.
     *
     * @param value - the value to reset the filter with
     */
    public void reset(A value);
```

An IdentityFilter is a generic class with one type parameter that returns the value input without modification. The output value has the same type as the input type.
The following is pseudo-code for the IdentityFilter class.

```
    /**
     * Filters by returning the identity of the input.
     *
     * @param value - the value to filter
     * @return the identity of the value
     */
    public A filter(A value){
        check that value is not null
        return value
    }
```

A ScalarFilter is an interface that only handles double-precision floating point numbers, so it is not generic.

A ScalarFilter is a filter that maintains a previously calculated double value and appends an input double value to that value in order to calculate the output desired for the implementing filter. The filter can be reset with a given value of type double.
The following are the method signatures of the ScalarFilter interface:

```
    /**
```

```
     * Filters the input value based on the previous input.
     * Returns the filtered value.
     *
     * @param value - the value to filter
     * @return the filtered value
     */
    public double filter(double value);

    /**
     * Resets the filter with the input value.
     *
     * @param value - the value to reset the filter with
     */
    public void reset(double value);
```

A MaxFilter is a generic Filter that filters the input data for the maximum value since initialization or the last reset. A comparator is used for comparing values in the MaxFilter. A MaxFilter can be reset with a value of type A which just resets the stored max value. The following is pseudo-code for the MaxFilter class:

```
    // The maximum value found thus far.
    A max;

    /**
     * Filters the maximum input value since the filter was initialized or
     * was last reset.
     *
     * @param value - the value to filter
     * @return the maximum seen by the filter thus far as type B
     */
    public B filter(A value) {
        check that value is not null

        if value is greater than max
            max <- value

        return max as type B
    }

    /**
     * Resets the filter with the specified value.
     *
     * @param value - the value to reset the filter with
     */
    public void reset(A value){
        max <- value
    }
```

A MinFilter is a generic Filter that filters the input data for the minimum value since initialization or the last reset. A comparator is used for comparing values in the MinFilter. A MinFilter can be reset with a value of type A which just resets the stored min value. The following is pseudo-code for the MinFilter class:

```
    // The minimum value found thus far.
    A min;

    /**
     * Filters the minimum input value since the filter was initialized or
     * was last reset.
     *
     * @param value - the value to filter
     * @return the minimum seen by the filter thus far as type B
     */
    public B filter(A value) {
        check that value is not null

        if value is less than min
            min <- value

        return min as type B
    }

    /**
     * Resets the filter with the specified value.
     *
     * @param value - the value to reset the filter with
     */
    public void reset(A value){
        min <- value
    }
```

A FilterN is a generic abstract class with one type parameter, A, that must extend a comparable class for comparison during filtering. This filter only considers the last N values input to the filter. The filter can be reset with a given value of type A which is added to the value list after it is cleared.

```
    // The count to reset the filter at.
    int n

    // An array of stored input values.
    List<A> values

    /**
     * Constructs a new Filter N that tracks n
     * previous inputs for filtering.
     *
     * @param n - the number of previous inputs tracked
     */
    public FilterN(int n){
        check that n is not null
        this.n <- n
        values <- a new array list
    }

    /**
     * Removes the front node of the list if
     * the size of the list is greater than or equal to
     * n.
     */
    void maintainN(){
        if size of values list >= n
            remove the front value from list

    }

    /**
     * Resets the filter with the given value.
     *
     * @param value - the value to reset the filter with
     */
    public void reset(A value){
        check that value is not null
        clear the value list
        add value to the value list
    }

    /**
     * Gets the list of values collected by this filter.
     *
     * @return the list of values collected by this filter
     */
    public List<A> getValues() {
        return values
    }
```

MaxFilterN is a generic FilterN of type A and Filter of type A and B that returns the maximum of the last N values seen or the maximum value seen since the last reset if less than N values were encountered.
The following is pseudo-code for the MaxFilterN class:

```
    /**
     * Constructs a Max Filter that resets after N
     * values are filtered.
     *
     * @param n - the number of calls to reset
     *          the filter at
     */
    public MaxFilterN(int n){
        call parent's constructor with n
    }

    /**
     * Filters the input value by returning the maximum
     * value input of the last N inputs or since the
     * last reset.
     *
     * @param value - the value to filter
     * @return the maximum value yet seen by the filter
     */
    public B filter(A value){
        check that value is not null
        call procedure maintainN
        add value to list of values
        return max of values
    }

    /**
     * Returns the maximum element in the list of values.
     *
     * @return the max element in the list of values
     */
    private B max(){
        A max <- null
        boolean first <- true
        for each value in list of values
            if first = true
                first <- false
                max <- value

            if value is greater than max
                max <- value

        return max as type B
    }
```

MinFilterN is a generic FilterN of type A and Filter of type A and B
that returns the minimum of the last N values seen or the minimum value seen since
the last reset if less than N values were encountered.
The following is pseudo-code for the MinFilterN class:

```
    /**
     * Constructs a Min Filter that resets after N
     * values are filtered.
     *
     * @param n - the number of calls to reset
     *          the filter at
     */
    public MinFilterN(int n){
        call parent's constructor with n
    }

    /**
     * Filters the input value by returning the minimum
     * value input of the last N inputs or since the
     * last reset.
     *
     * @param value - the value to filter
     * @return the minimum value yet seen by the filter
     */
    public B filter(A value){
        check that value is not null
        call procedure maintainN
        add value to list of values
        return min of values
    }

    /**
     * Returns the minimum element in the list of values.
     *
     * @return the min element in the list of values
     */
    private B min(){
        A min <- null
        boolean first <- true
        for each value in list of values
            if first = true
                first <- false
                min <- value

            if value is less than min
                min <- value

        return min as type B
    }
```

An AveragingFilter is a ScalarFilter that returns the average of the input data

since the filter was initialized or last reset.
The following is pseudo-code for the AveragingFilter class:

```
    // The sum of the previously entered values.
    double sum;

    // The count of entered values.
    int count;

    /**
     * Constructs an Averaging Filter.
     */
    public AveragingFilter(){
        sum <- 0
        count <- 0
    }

    /**
     * Calculates the average of the data entered thus far
     *
     * @param value - the value to filter
     * @return the average of the filter input seen thus far
     */
    public double filter(double value){
        check that value is not null
        sum <- sum + value
        count <- count + 1
        return sum / count
    }

    /**
     * Resets the filter with the given value.
     *
     * @param value - the value to reset the filter with
     */
    public void reset(double value){
        check that value is not null
        sum <- value
        count <- 0
    }
```

An AveragingFilterN is a ScalarFilter and a FilterN of type Double.
This filter returns the average of the input data since the last N values
or the last reset.
The following is pseudo-code for the AveragingFilterN class:

```
    /**
     * Constructs an Averaging Filter that resets after N
     * values are filtered.
     *
     * @param n - the number of calls to reset
     *          the filter at
     */
    public AveragingFilterN(int n){
        call parent's constructor with n
    }

    /**
     * Calculates the average of the data entered thus far
     *
     * @param value - the value to filter
     * @return the average of the last N values or since the last reset
     */
    public double filter(double value){
        check that value is not null
        call procedure maintainN
        add value to list of values
        return average of values
    }

    /**
     * Calculates the average of the stored values.
     *
     * @return the average of the stored values
     */
    private double average() {
        double sum <- 0
        for each value in list of values
            sum <- sum + value

        return sum / size of values list
    }

    /**
     * Resets the filter with the given value.
     *
     * @param value - the value to reset the filter with
     */
    public void reset(double value){
        check that value is not null
        clear values list
        add value to the values list
    }
```

A FilterCascade is a generic Filter of types A and B that is a cascade of different filters of types A and B which filter a given input value through each filter sequentially and return the filtered output from the cascade as type B. Filters in the filter cascade can only be reset individually but not as a whole.
The following is pseudo-code for the FilterCascade class:

```
    // The filters of the filter cascade.
    List<Filter<A, B>> filters

    /**
     * Constructs a filter cascade from generically typed filters.
     *
     * @param filters - the filters to build the filter cascade with
     */
    public FilterCascade(List<Filter<A, B>> filters){
        this.filters <- filters
    }

    /**
     * Filters the input value through the filter cascade.
     *
     * @param value - the value to filter
     * @return the filtered value
     */
    public B filter(A value){
        A output <- value

        for each filter filter in list of filters
            output <- filter.filter(output) as type A

        return output as type B
    }

    /**
     * Resets the selected filter with the given value.
     *
     * @param filter - the filter to reset at the given index
     * @param value - the value to reset the filter with
     */
    public void reset(int filter, A value){
        check that filter and value are not null
        check 0 <= value < size of filter list
        reset the filter in the filter list with value
    }

    /**
     * Does nothing in this implementation.
     *
     * @param value - the value to reset the filter with
     */
    public void reset(Comparable value) {
    }
```

A ScalarLinearFilter is a ScalarFilter that filters input based on a linear equation (y(i) + a(1)y(i-1)+...+a(M)y(i-M) = b(0)x(i)+...+b(N)x(i-N)) which considers the current iteration of input, previous input values and the current input value. The filter can be reset with a given value of type double.
The following is pseudo-code for the ScalarLinearFilter class:

```
    // The output boundary coefficient for the linear equation.
    int M

    // The input boundary coefficient for the linear equation.
    int N

    // The current iteration of the filter.
    int i

    // The sum of the input.
    double inputSum

    // The sum of the output.
    double outputSum

    // The output multiplier list.
    List<Double> a

    // The input multiplier list.
    List<Double> b

    // The previous input list.
    List<Double> x
```

```
// The previous output list.
List<Double> y

/**
 * Constructs a scalar linear filter with boundary coefficients N and M
 * and lists of multipliers a and b for input and output consecutively.
 *
 * @param M - the output boundary coefficient
 * @param N - the input boundary coefficient
 * @param a - the multiplier list for output
 * @param b - the multiplier list for input
 */
public ScalarLinearFilter(int M, int N, ArrayList<Double> a, ArrayList<Double>
   b){
    this.M <- M
    this.N <- N
    this.a <- a
    this.b <- b
    this.i <- 0
    this.inputSum <- 0
    this.outputSum <- 0
    x <- new array list initialized with M zeroes
    y <- new array list initialized with N zeroes
}

/**
 * Filters the input by calculating the output at
 * the current iteration given the linear equation
 * for the scalar linear filter.
 *
 * @param in - the input value to calculate the output in
 *           relation to
 * @return the output value y(i) of the linear equation solution
 */
public double filter(double in) {
    check that in is not null
    add in to x at index i
    double out <- call function sumInput - call function sumOutput
    add out to y at index i
    i <- i + 1
    return out
}

/**
 * Calculates the right (input) side of the linear equation.
 * Sum of b(n) * x(i-n), where n starts at 0 and ends at N
 * and i is the current iteration of filter input.
 *
 * @return the sum of the input side of the linear equation
 */
private double sumInput() {
    check that b and x are not empty
    check that b has n elements

    double sum <- 0
    label IN_SUM_LOOP:
    for n <- 0 to N do
        //sum = sum + b(n) * x(i-n)
        if i - n >= 0
```

```
            sum <- sum + value of b at index n * value of x at index i - n
        else
            //will only add zeroes to sum, so break
            break from IN_SUM_LOOP

    return sum + inputSum
}

/**
 * Calculates the left (output) side of the linear equation without adding the
 * output, y(i), of the current iteration i.
 * Sum of a(m) * y(i-m), where m starts at 1 and ends at M.
 *
 * @return the left side of the linear equation without the output included
 */
private double sumOutput() {
    check that a and y are not empty
    check that a has m elements

    double sum <- 0
    label OUT_SUM_LOOP:
    for m <- 1 to M do
        //sum = sum + a(m) * y(i-m)
        if i - m >= 0
            sum <- sum + value of a at index m * value of y at index i - m
        else
            //will only add zeroes to sum, so break
            break from OUT_SUM_LOOP

    return sum + outputSum
}

/**
 * Resets the filter with the given value r.
 * Sets the record of previous input value to r.
 * Sets the record of previous output value to
 * r(sum of b(0)-b(N)) / (1 + sum of a(1) - a(M)).
 *
 * @param r - the value to reset the filter with
 */
public void reset(double r) {
    double dividend <- 0
    double quotient <- 1
    for n <- 0 to N do
        dividend <- dividend + value of b at index n

    for n <- 1 to M do
        quotient <- quotient + value of a at index m

    i <- 0
    inputSum <- r
    dividend <- r * dividend
    outputSum <- dividend / quotient
}

/**
 * Gets the input boundary coefficient.
 *
 * @return the input boundary coefficient
```

```
     */
    public int getN() {
        return N
    }

    /**
     * Gets the list of input multipliers.
     *
     * @return the list of input multipliers
     */
    public List<Double> getB() {
        return b
    }
```

A FIRFilter is a ScalarLinearFilter with a(0,1) and M initialized to 0. Summation of the output portion of the linear equation of a scalar linear filter is ignored in a FIR Filter. The following is pseudo-code for the FIRFilter class:

```
    /**
     * Constructs a scalar linear filter with input boundary coefficient N
     * and list of multipliers b for input.
     * The list of multipliers for the output, a, are all initialized to 0.
     * The boundary coefficient for the output, M, is set to 0 since summation
     * of output does not take place in a FIR filter.
     *
     * @param N - the input boundary coefficient
     * @param b - the multiplier list for input
     */
    public FIRFilter(int N, ArrayList<Double> b) {
        call parent's constructor with M <- 0, N <- N, a <- an array list with two
        zeroes, b <- b
    }
```

A GainFilter is a FIRFilter that multiplies only the input by a constant factor, b(0), to get the filtered output.

```
    /**
     * Constructs a FIR Filter with input coefficient as 0
     * and b(0) as the gain factor, the only factor in the list.
     *
     * @param b - the gain factor of the filter
     */
    public GainFilter(double b) {
        call parent's constructor with N <- 0, b <- b as an array list
    }
```

A Binomial Filter is a FIR Filter where b(i) <- (nCi), which is the binomial coefficient at i, where i is the current iteration of the input.

```
    /**
     * Constructs a scalar linear filter with input boundary coefficient N
     * and list of multipliers b for input.
     * The list of multipliers for the output, a, are all initialized to 0.
     * The boundary coefficient for the output, M, is set to 0 since summation
     * of output does not take place in a FIR filter.
     *
     * @param N - the input boundary coefficient
     * @param b - the multiplier list for input
     */
    public BinomialFilter(int N, ArrayList<Double> b) {
        call parent's constructor with N <- N, b <- b
        call procedure setBinomials
    }

    /**
     * Sets b(i) to N choose i for each b in the list of
     * input multipliers where i ranges from 0 to N.
     */
    private void setBinomials(){
        for i <- 0 to N
            double b <- call function binomialC with N and i
            set value of list B at index i to value b
        }
    }

    /**
     * Calculates the binomial coefficient based on the input numbers.
     *
     * @param n - the number to choose from
     * @param i - the number that can be chosen
     * @return the binomial coefficient n C i
     */
    private int binomialC(int n, int i){
        if i = 0 or i = n
            return 1

        return (call binomialC with n-1 and i-1) + (call binomialC with n-1 and i)
    }
```

In instances where reset is not supported given the documentation, the reset procedure will not do anything. In these cases, the method will simply be empty. Thus, the method will be callable but it will not actually alter the filter.

Error-handling will be provided by a type of validator class that will check for any bad input (i.e. null) and boundaries (i.e. coefficients in ScalarLinearFilter). This will provide a type of barrier from bad input. Currently, each method is set up to handle errors individually. Thus, if a method finds an error in input via the validator, an exception will be thrown recognizing the error. This also goes for the way the filter was constructed. If the filter was constructed poorly with incorrect types, etc., then the validator will throw an exception recognizing the error that took place.

For testing public methods, I will create a test class for each filter that tests the filter with diverse input. I will test for good data, bad data, data flow, structured-basis, boundary, complex boundary, and stress. If any of the tests are not applicable, I will leave them out (i.e. complex boundary).

For testing private methods, I will write inner test classes for each of the filters that will be invoked by the test harness while testing for the same principles as noted above for the pubic methods.