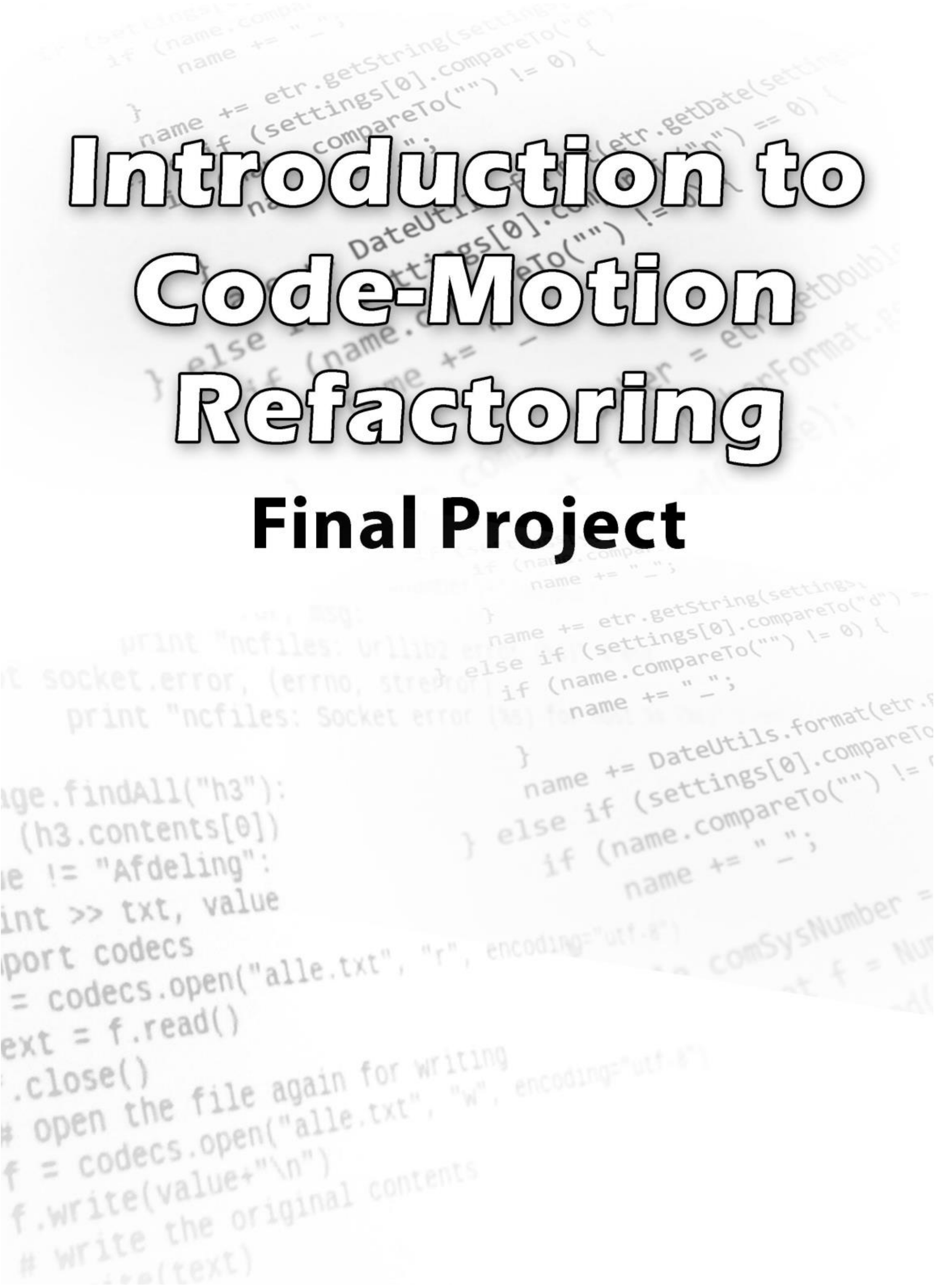
Introduction to Code-Motion Refactoring

Final Project



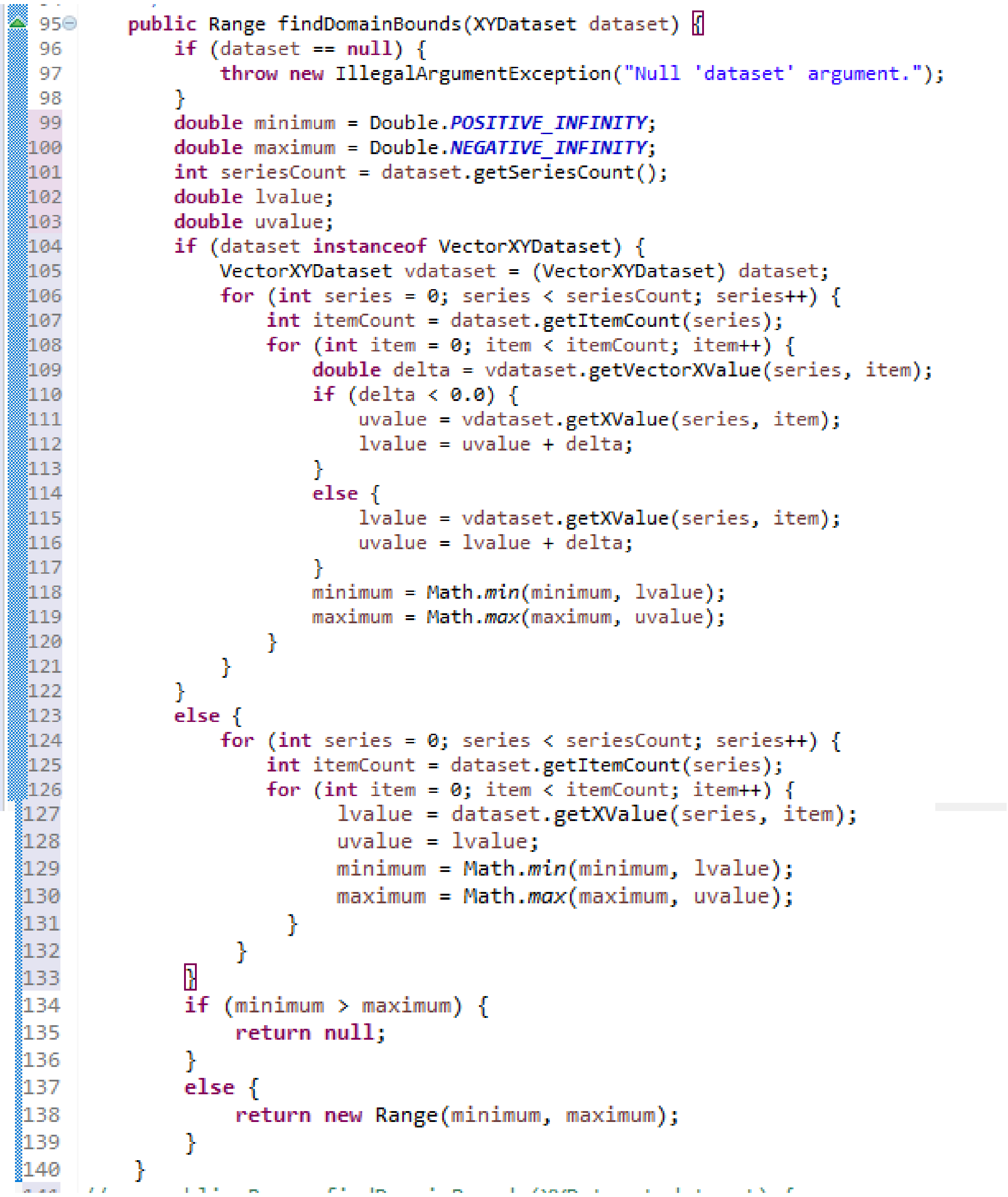
Sliding:

We chose a method called "findRangBounds" from an open source project which can be found at:

https://github.com/alwaqfi/jfreechart-1.0.10/blob/master/src/org/jfree/chart/renderer/xy/VectorRenderer.java

jfreechart-1.0.10/src/org/jfree/chart/renderer/xy/VectorRenderer.java

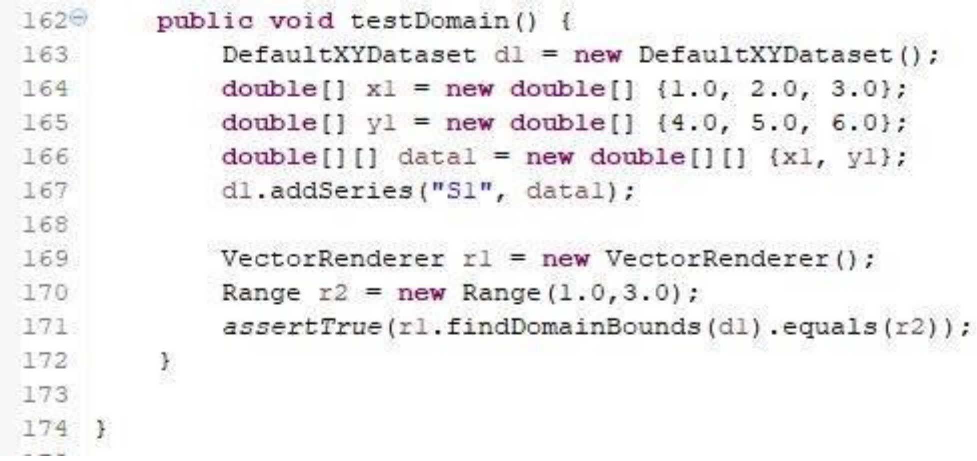
original method:



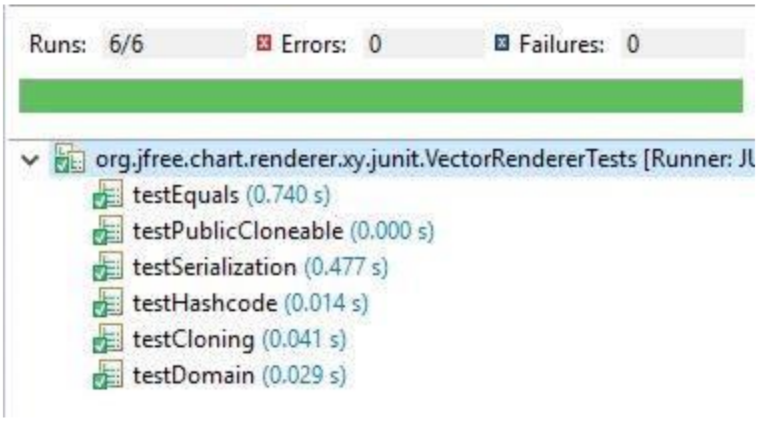
```
    public Range findDomainBounds(XYDataset dataset) {
        if (dataset == null) {
            throw new IllegalArgumentException("Null 'dataset' argument.");
        }
        double minimum = Double.POSITIVE_INFINITY;
        double maximum = Double.NEGATIVE_INFINITY;
        int seriesCount = dataset.getSeriesCount();
        double lvalue;
        double uvalue;
        if (dataset instanceof VectorXYDataset) {
            VectorXYDataset vdataset = (VectorXYDataset) dataset;
            for (int series = 0; series < seriesCount; series++) {
                int itemCount = dataset.getItemCount(series);
                for (int item = 0; item < itemCount; item++) {
                    double delta = vdataset.getVectorXValue(series, item);
                    if (delta < 0.0) {
                        uvalue = vdataset.getXValue(series, item);
                        lvalue = uvalue + delta;
                    }
                    else {
                        lvalue = vdataset.getXValue(series, item);
                        uvalue = lvalue + delta;
                    }
                    minimum = Math.min(minimum, lvalue);
                    maximum = Math.max(maximum, uvalue);
                }
            }
        }
        else {
            for (int series = 0; series < seriesCount; series++) {
                int itemCount = dataset.getItemCount(series);
                for (int item = 0; item < itemCount; item++) {
                    lvalue = dataset.getXValue(series, item);
                    uvalue = lvalue;
                    minimum = Math.min(minimum, lvalue);
                    maximum = Math.max(maximum, uvalue);
                }
            }
        }
        if (minimum > maximum) {
            return null;
        }
        else {
            return new Range(minimum, maximum);
        }
    }
```

Since this method doesn't has a test we created another method for testing, please add the method in here:

jfreechart-1.0.10/tests/org/jfree/chart/renderer/xy/junit/**VectorRendererTests.java**



```
    public void testDomain() {
        DefaultXYDataset d1 = new DefaultXYDataset();
        double[] x1 = new double[] {1.0, 2.0, 3.0};
        double[] y1 = new double[] {4.0, 5.0, 6.0};
        double[][] data1 = new double[][] {x1, y1};
        d1.addSeries("S1", data1);

        VectorRenderer r1 = new VectorRenderer();
        Range r2 = new Range(1.0,3.0);
        assertTrue(r1.findDomainBounds(d1).equals(r2));
    }

}
```



```
Runs: 6/6      ☒ Errors: 0      ☒ Failures: 0

org.jfree.chart.renderer.xy.junit.VectorRendererTests [Runner: JU
    testEquals (0.740 s)
    testPublicCloneable (0.000 s)
    testSerialization (0.477 s)
    testHashcode (0.014 s)
    testCloning (0.041 s)
    testDomain (0.029 s)
```

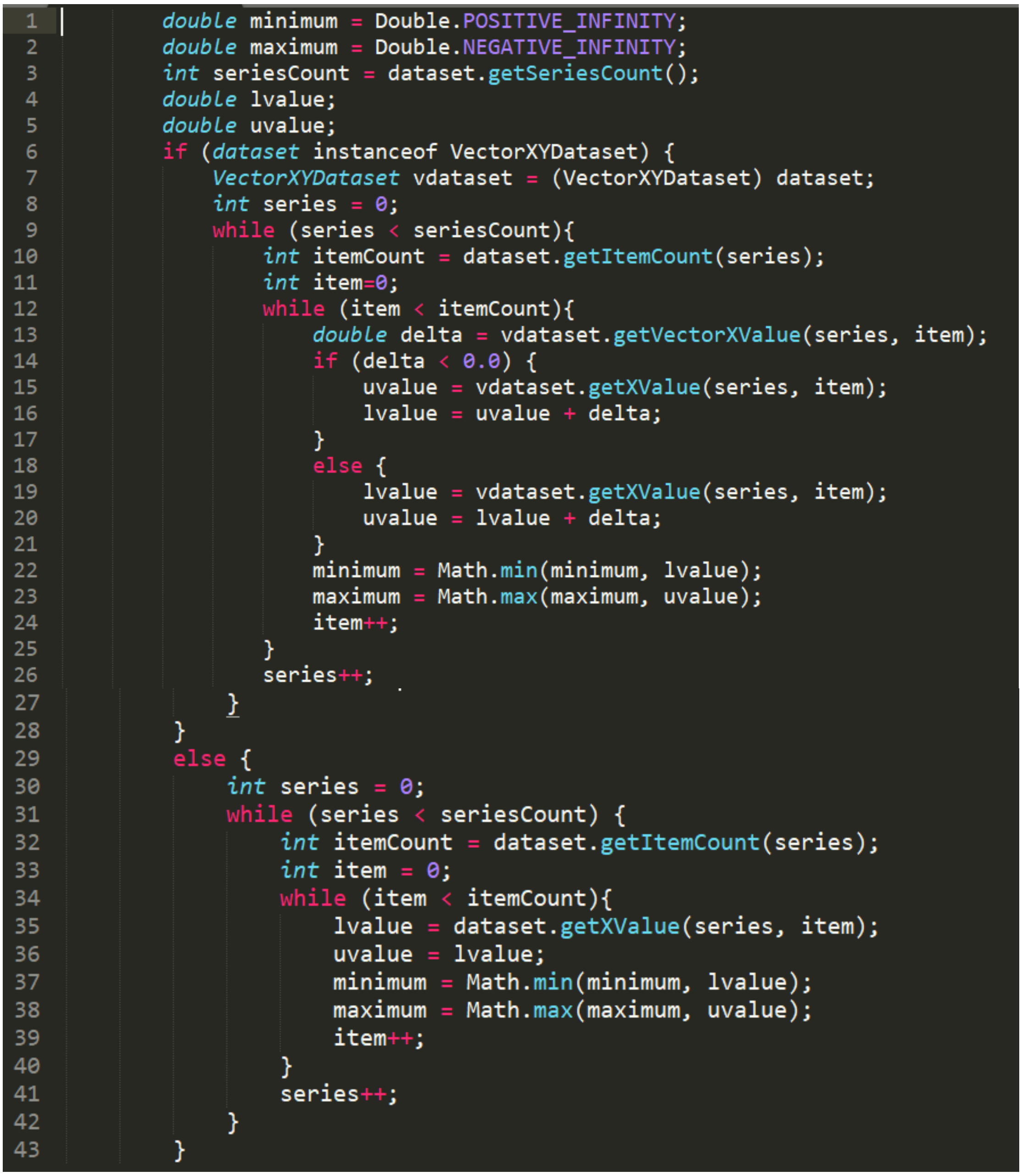
Step1:

we focused on a specific block of code in the method, and applied the Sliding algorithm on that specific block of code.

This block of code compute tow variables independently {maximum, minimum}.

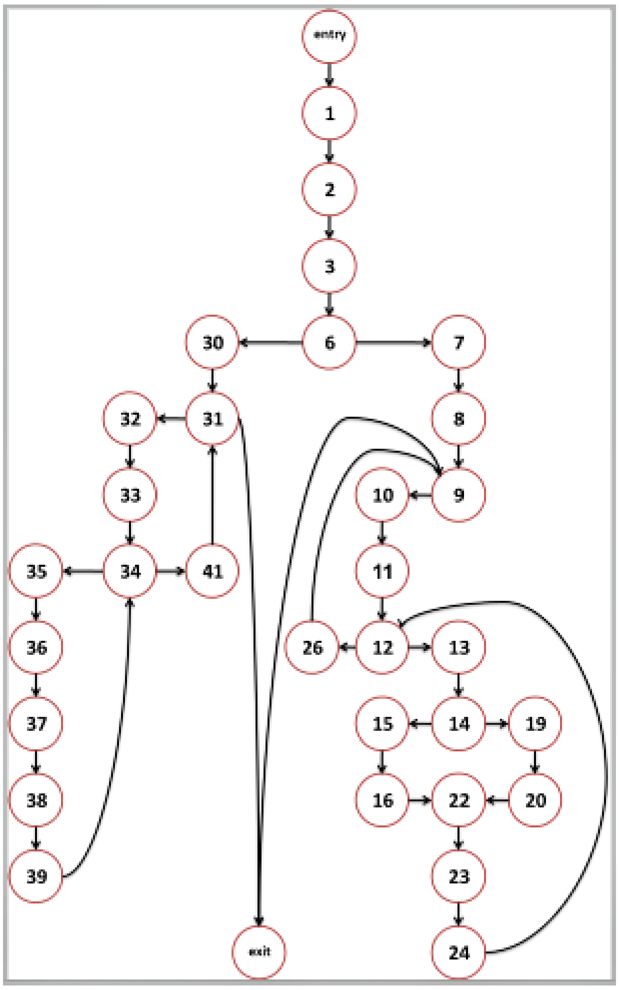
We would like to split those computations into tow sperate methods.

We chose the block of code from lines 99-133, for convenience we numbered those lines from 1-43 after converting "for" loops to "while" loops.



```
        double minimum = Double.POSITIVE_INFINITY;
        double maximum = Double.NEGATIVE_INFINITY;
        int seriesCount = dataset.getSeriesCount();
        double lvalue;
        double uvalue;
        if (dataset instanceof VectorXYDataset) {
            VectorXYDataset vdataset = (VectorXYDataset) dataset;
            int series = 0;
            while (series < seriesCount){
                int itemCount = dataset.getItemCount(series);
                int item=0;
                while (item < itemCount){
                    double delta = vdataset.getVectorXValue(series, item);
                    if (delta < 0.0) {
                        uvalue = vdataset.getXValue(series, item);
                        lvalue = uvalue + delta;
                    }
                    else {
                        lvalue = vdataset.getXValue(series, item);
                        uvalue = lvalue + delta;
                    }
                    minimum = Math.min(minimum, lvalue);
                    maximum = Math.max(maximum, uvalue);
                    item++;
                }
                series++;
            }
        }
        else {
            int series = 0;
            while (series < seriesCount) {
                int itemCount = dataset.getItemCount(series);
                int item = 0;
                while (item < itemCount){
                    lvalue = dataset.getXValue(series, item);
                    uvalue = lvalue;
                    minimum = Math.min(minimum, lvalue);
                    maximum = Math.max(maximum, uvalue);
                    item++;
                }
                series++;
            }
        }
```

Build the CFG:



Build the PDG:

Edge	type
(Entry,1)	**Control**
(Entry,2)	**Control**
(Entry,3)	**Control**
(Entry,6)	**Control**
(6,7)	**Control**
(6,8)	**Control**
(6,9)	**Control**
(6,30)	**Control**
(6,31)	**Control**
(9,10)	**Control**
(9,11)	**Control**
(9,12)	**Control**
(9,26)	**Control**
(12,13)	**Control**
(12,14)	**Control**
(12,22)	**Control**
(12,23)	**Control**
(12,24)	**Control**
(14,15)	**Control**
(14,16)	**Control**
(14,19)	**Control**
(14,20)	**Control**
(31,32)	**Control**
(31,33)	**Control**
(31,34)	**Control**
(31,41)	**Control**
(34,35)	**Control**
(34,36)	**Control**
(34,37)	**Control**
(34,38)	**Control**
(34,39)	**Control**
(9, exit)	**Control**
(31, exit)	**Control**

Edge	type	Vars
(1,22)	Flow	**{minimum}**
(1,37)	Flow	**{minimum}**
(2,23)	Flow	**{maximum}**
(2,38)	Flow	**{maximum}**
(3,31)	Flow	**{seriesCount}**
(3,9)	Flow	**{seriesCount}**
(8,9)	Flow	**{series}**
(8,10)	Flow	**{series}**
(8,13)	Flow	**{series}**
(8,15)	Flow	**{series}**
(8,19)	Flow	**{series}**
(8,26)	Flow	**{series}**
(26,9)	Flow	**{series}**
(26,10)	Flow	**{series}**
(26,13)	Flow	**{series}**
(26,15)	Flow	**{series}**
(26,19)	Flow	**{series}**
(26,26)	Flow	**{series}**
(30,31)	Flow	**{series}**
(30,32)	Flow	**{series}**
(30,35)	Flow	**{series}**
(30,41)	Flow	**{series}**
(41,32)	Flow	**{series}**
(41,35)	Flow	**{series}**
(41,41)	Flow	**{series}**
(41,31)	Flow	**{series}**
(16,22)	Flow	**{lvalue}**
(19,22)	Flow	**{lvalue}**
(19,20)	Flow	**{lvalue}**
(35,36)	Flow	**{lvalue}**
(35,37)	Flow	**{lvalue}**
(34,38)	Flow	**{uvalue}**
(15,16)	Flow	**{uvalue}**

Edge	type	Vars
(1, exit)	Flow	**{minimum}**
(22, exit)	Flow	**{minimum}**
(37, exit)	Flow	**{minimum}**
(2, exit)	Flow	**{maximum}**
(23, exit)	Flow	**{maximum}**
(38, exit)	Flow	**{maximum}**

Edge	type	Vars
(15,23)	Flow	**{uvalue}**
(20,23)	Flow	**{uvalue}**
(36,38)	Flow	**{uvalue}**
(10,12)	Flow	**{itemCount}**
(32,34)	Flow	**{itemCount}**
(32,34)	Flow	**{item}**
(11,12)	Flow	**{item}**
(11,13)	Flow	**{item}**
(11,15)	Flow	**{item}**
(11,19)	Flow	**{item}**
(11,24)	Flow	**{item}**
(24,12)	Flow	**{item}**
(24,13)	Flow	**{item}**
(24,15)	Flow	**{item}**
(24,19)	Flow	**{item}**
(24,24)	Flow	**{item}**
(33,34)	Flow	**{item}**
(33,35)	Flow	**{item}**
(33,39)	Flow	**{item}**
(39,34)	Flow	**{item}**
(39,35)	Flow	**{item}**
(39,39)	Flow	**{item}**
(13,14)	Flow	**{delta}**
(13,16)	Flow	**{delta}**
(13,20)	Flow	**{delta}**
(entry,3)	Flow	**{dataset}**
(entry,6)	Flow	**{dataset}**
(entry,7)	Flow	**{dataset}**
(entry,10)	Flow	**{dataset}**
(entry,32)	Flow	**{dataset}**
(entry,35)	Flow	**{dataset}**
(7,13)	Flow	**{ vdataset }**
(7,15)	Flow	**{ vdataset }**

Edge	type	Vars
(16,13)	anti	**{delta}**
(20,13)	anti	**{delta}**
(20,16)	anti	**{lvalue}**
(20,19)	anti	**{lvalue}**
(22,16)	anti	**{lvalue}**
(22,19)	anti	**{ lvalue }**
(36,35)	anti	**{ lvalue }**
(37,35)	anti	**{ lvalue }**
(16,20)	anti	**{uvalue}**
(16,15)	anti	**{uvalue}**
(23,15)	anti	**{uvalue}**
(23,20)	anti	**{uvalue}**
(38,36)	anti	**{uvalue}**

Edge	type	Vars
(23,23)	anti	**{maximum}**
(38,38)	anti	**{ maximum }**
(22,22)	anti	**{minimum}**
(37,37)	anti	**{ minimum }**
(9,26)	anti	**{ series }**
(10,26)	anti	**{ series }**
(13,26)	anti	**{ series }**
(15,26)	anti	**{ series }**
(19,26)	anti	**{ series }**
(26,26)	anti	**{ series }**
(31,41)	anti	**{ series }**
(32,41)	anti	**{series }**
(35,41)	anti	**{series }**
(41,41)	anti	**{series}**
(12,10)	anti	**{itemCount}**
(34,32)	anti	**{itemCount}**
(12,24)	anti	**{item}**
(15,24)	anti	**{item}**
(13,24)	anti	**{item}**
(19,24)	anti	**{item}**
(24,24)	anti	**{item}**
(34,39)	anti	**{item}**
(35,39)	anti	**{item}**
(39,39)	anti	**{item}**
(14,13)	anti	**{delta}**

Step 2:

We apply sliding algorithm on V={minimum}.

Slice:

To do the slice we remove flow dependencies to exit of any variable other than "minimum".

Now we do back tracking starting from exit node up, using control and flow dependencies.

Slice(exit) = {entry, 1, 3, 6, 7, 8, 9, 10, 11, 12, 13, 14 ,15, 16, 19, 22, 24, 26, 30, 31, 32, 33, 34, 35, 37, 39, 41, exit}

Co- Slice:

To do the Co-Slice we remove flow dependencies created by the variable "minimum" (y, x) such that there is no anti dependence (x, z) created by the variable "minimum".

Those are the edges that we remove:

(1, exit)	Flow	{minimum}
(22, exit)	Flow	{minimum}
(37, exit)	Flow	{minimum}

Co-Slice(exit) = {entry, 2, 3, 6, 7, 8, 9, 10, 11, 12, 13, 14 ,15, 19, 20, 23, 24, 26, 30, 31, 32, 33, 34, 35, 36, 38, 39 ,41, exit}

Compensations checking:

Pen1: The value of the extracted variable could change in the Co-Slice.

There is no definition to variable "minimum" in any Co-Slice's nodes, therefore **pen1 = {}**.

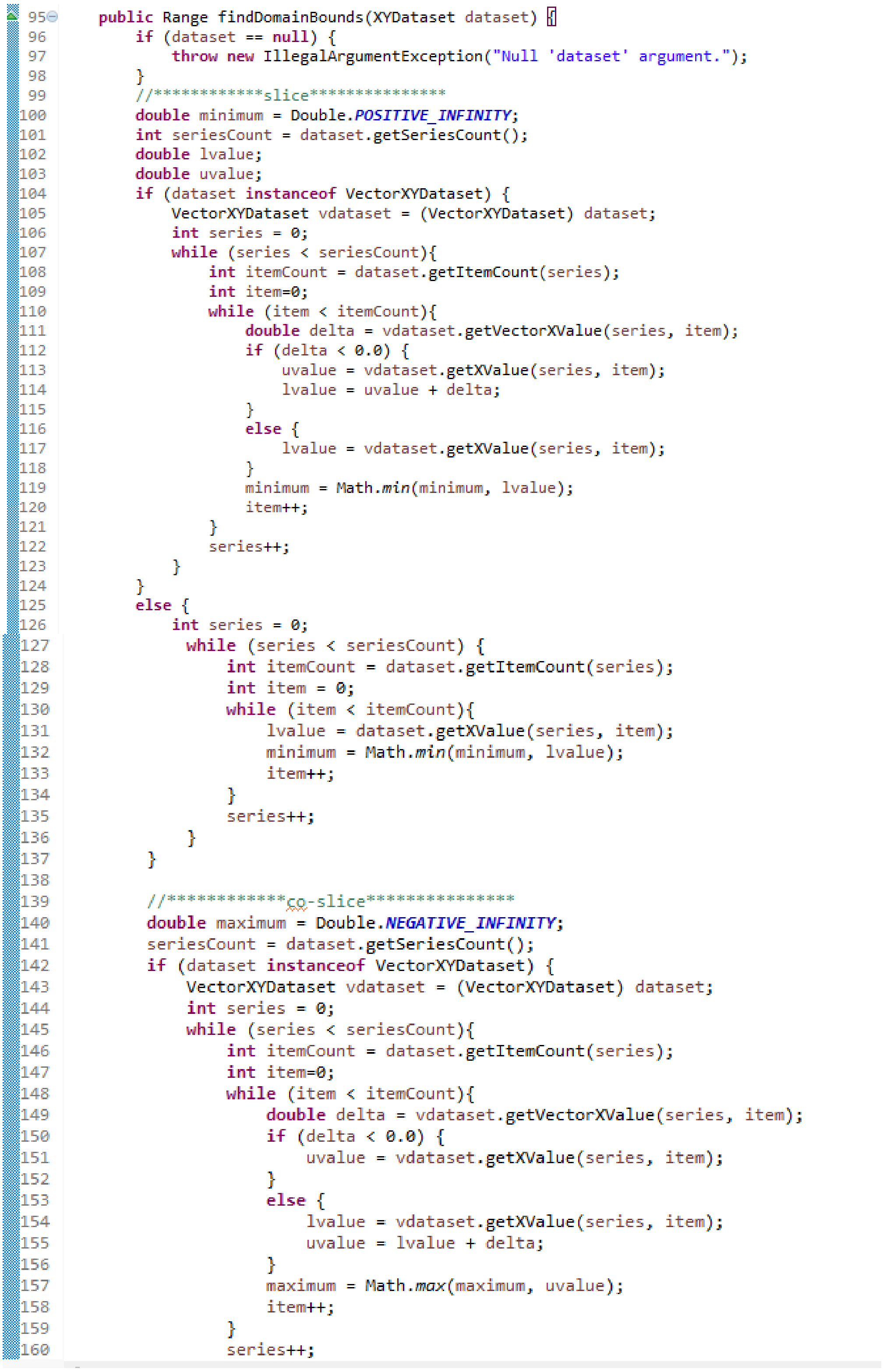
Pen2: there exists a usage in the Co-Slice of a non-final value of the extracted variable.

There is no usage to variable "minimum" in any Co-Slice's nodes, therefore **pen2 = {}**.

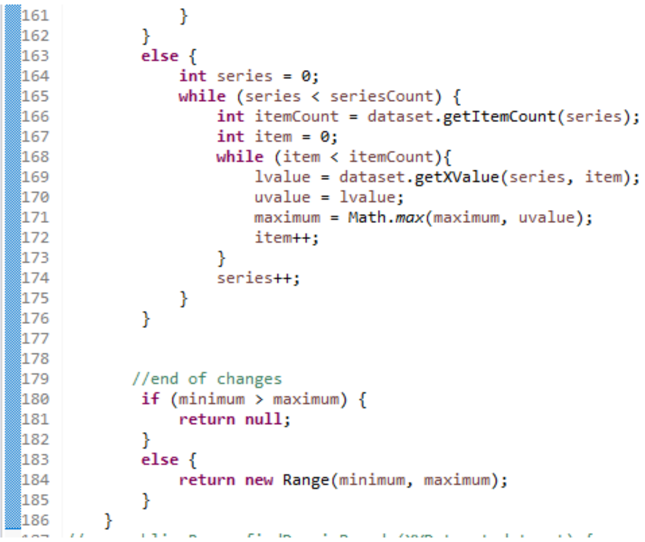
Pen3: the slice could change the value of unextracted variable, that its initial value is required at the co-slice.

There isn't Slice's nodes defining variables causing flow dependencies from entry to Co-Slice's nodes, therefore **pen3 = {}**.

The result will be:



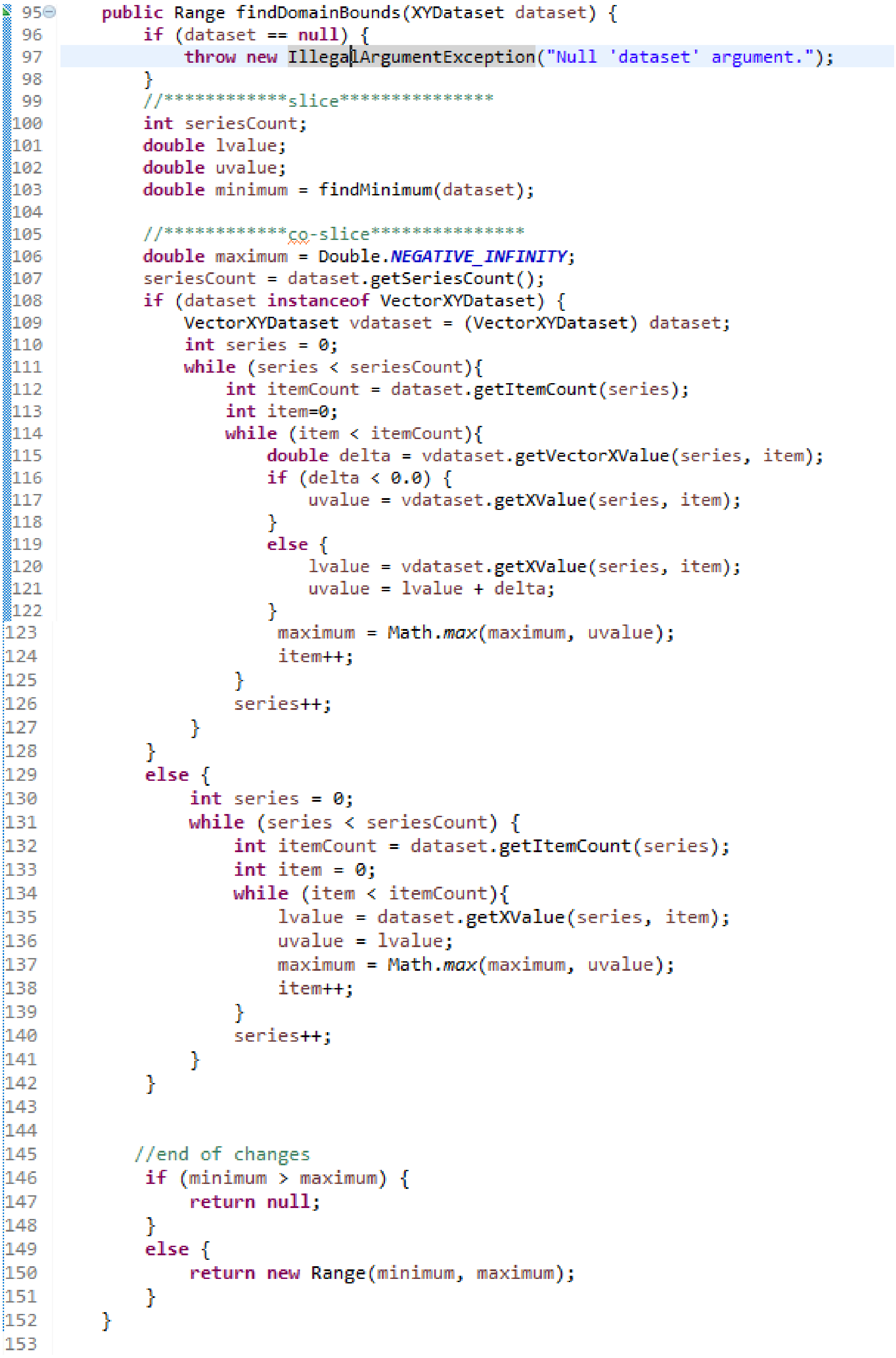
```
public Range findDomainBounds(XYDataset dataset) {
    if (dataset == null) {
        throw new IllegalArgumentException("Null 'dataset' argument.");
    }
    //************slice**************
    double minimum = Double.POSITIVE_INFINITY;
    int seriesCount = dataset.getSeriesCount();
    double lvalue;
    double uvalue;
    if (dataset instanceof VectorXYDataset) {
        VectorXYDataset vdataset = (VectorXYDataset) dataset;
        int series = 0;
        while (series < seriesCount){
            int itemCount = dataset.getItemCount(series);
            int item=0;
            while (item < itemCount){
                double delta = vdataset.getVectorXValue(series, item);
                if (delta < 0.0) {
                    uvalue = vdataset.getXValue(series, item);
                    lvalue = uvalue + delta;
                }
                else {
                    lvalue = vdataset.getXValue(series, item);
                }
                minimum = Math.min(minimum, lvalue);
                item++;
            }
            series++;
        }
    }
    else {
        int series = 0;
        while (series < seriesCount) {
            int itemCount = dataset.getItemCount(series);
            int item = 0;
            while (item < itemCount){
                lvalue = dataset.getXValue(series, item);
                minimum = Math.min(minimum, lvalue);
                item++;
            }
            series++;
        }
    }

    //************co-slice**************
    double maximum = Double.NEGATIVE_INFINITY;
    seriesCount = dataset.getSeriesCount();
    if (dataset instanceof VectorXYDataset) {
        VectorXYDataset vdataset = (VectorXYDataset) dataset;
        int series = 0;
        while (series < seriesCount){
            int itemCount = dataset.getItemCount(series);
            int item=0;
            while (item < itemCount){
                double delta = vdataset.getVectorXValue(series, item);
                if (delta < 0.0) {
                    uvalue = vdataset.getXValue(series, item);
                }
                else {
                    lvalue = vdataset.getXValue(series, item);
                    uvalue = lvalue + delta;
                }
                maximum = Math.max(maximum, uvalue);
                item++;
            }
            series++;
```



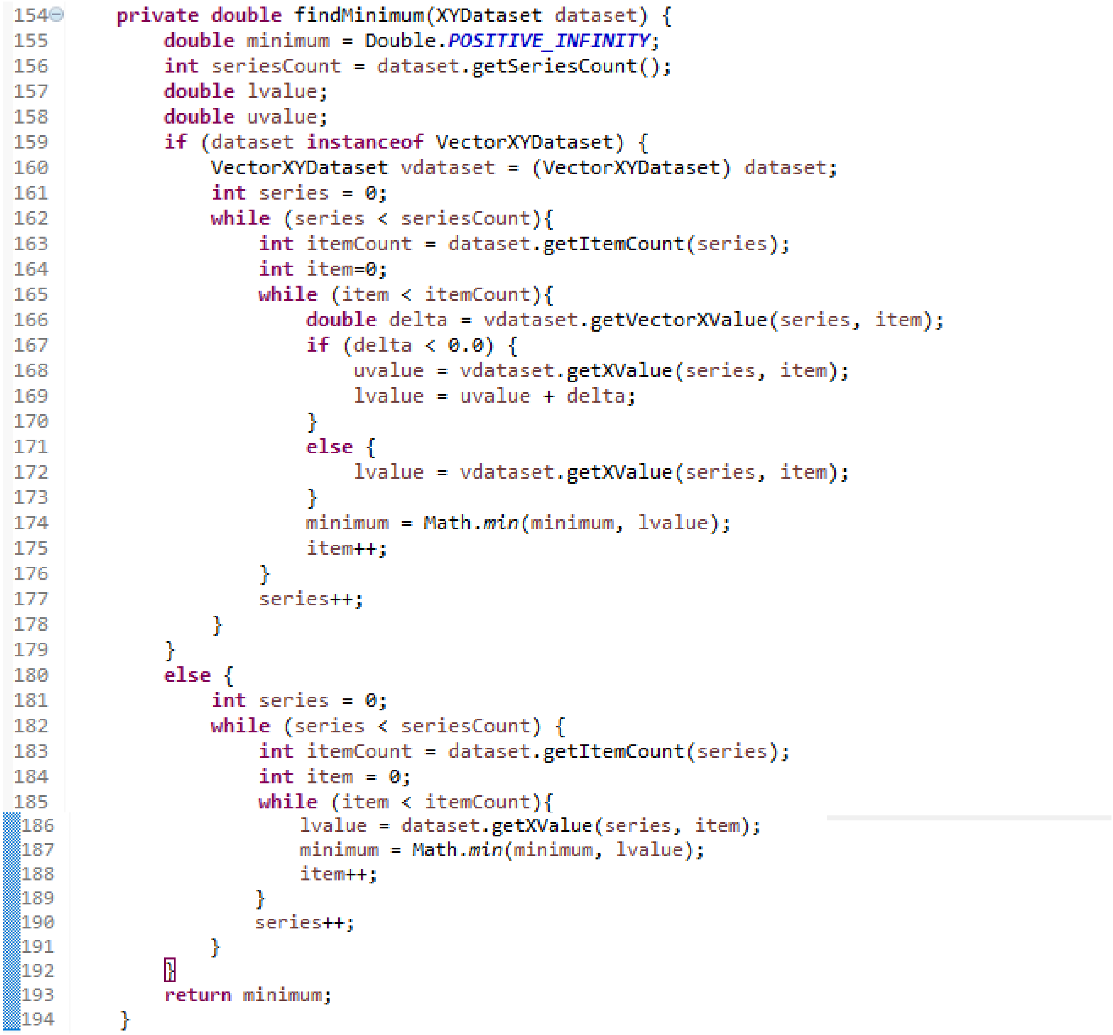
```
            }
        }
        else {
            int series = 0;
            while (series < seriesCount) {
                int itemCount = dataset.getItemCount(series);
                int item = 0;
                while (item < itemCount){
                    lvalue = dataset.getXValue(series, item);
                    uvalue = lvalue;
                    maximum = Math.max(maximum, uvalue);
                    item++;
                }
                series++;
            }
        }


       //end of changes
        if (minimum > maximum) {
            return null;
        }
        else {
            return new Range(minimum, maximum);
        }
    }
//    public Range findDomainBounds(XYDataset dataset) {
```

Step 3:

Extract the Slice into a new method.



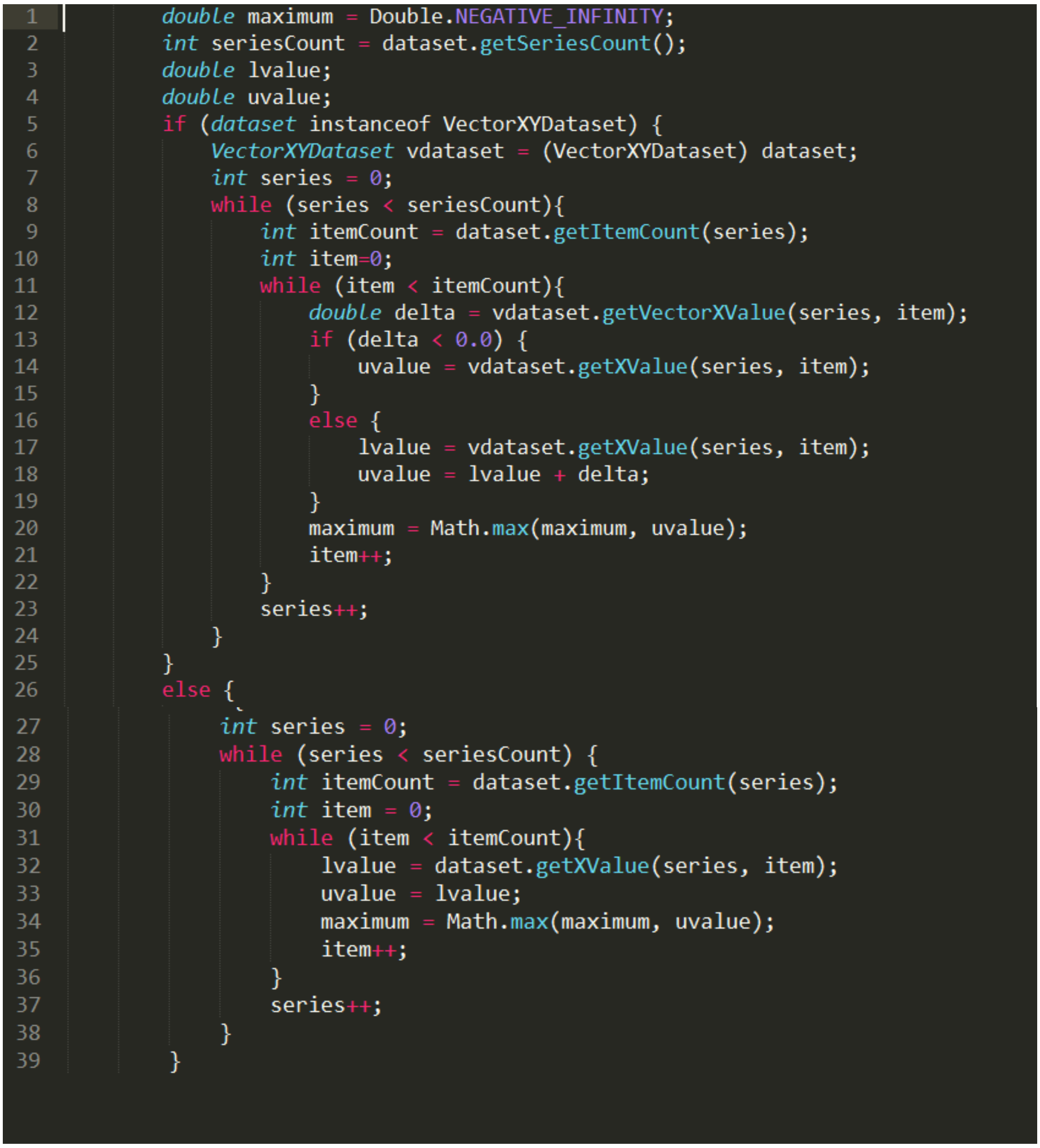
```
public Range findDomainBounds(XYDataset dataset) {
    if (dataset == null) {
        throw new IllegalArgumentException("Null 'dataset' argument.");
    }
    //************slice***************
    int seriesCount;
    double lvalue;
    double uvalue;
    double minimum = findMinimum(dataset);

    //************co-slice***************
    double maximum = Double.NEGATIVE_INFINITY;
    seriesCount = dataset.getSeriesCount();
    if (dataset instanceof VectorXYDataset) {
        VectorXYDataset vdataset = (VectorXYDataset) dataset;
        int series = 0;
        while (series < seriesCount){
            int itemCount = dataset.getItemCount(series);
            int item=0;
            while (item < itemCount){
                double delta = vdataset.getVectorXValue(series, item);
                if (delta < 0.0) {
                    uvalue = vdataset.getXValue(series, item);
                }
                else {
                    lvalue = vdataset.getXValue(series, item);
                    uvalue = lvalue + delta;
                }
                 maximum = Math.max(maximum, uvalue);
                 item++;
            }
            series++;
        }
    }
    else {
        int series = 0;
        while (series < seriesCount) {
            int itemCount = dataset.getItemCount(series);
            int item = 0;
            while (item < itemCount){
                lvalue = dataset.getXValue(series, item);
                uvalue = lvalue;
                maximum = Math.max(maximum, uvalue);
                item++;
            }
            series++;
        }
    }


   //end of changes
    if (minimum > maximum) {
        return null;
    }
    else {
        return new Range(minimum, maximum);
    }
}
```



```
    private double findMinimum(XYDataset dataset) {
        double minimum = Double.POSITIVE_INFINITY;
        int seriesCount = dataset.getSeriesCount();
        double lvalue;
        double uvalue;
        if (dataset instanceof VectorXYDataset) {
            VectorXYDataset vdataset = (VectorXYDataset) dataset;
            int series = 0;
            while (series < seriesCount){
                int itemCount = dataset.getItemCount(series);
                int item=0;
                while (item < itemCount){
                    double delta = vdataset.getVectorXValue(series, item);
                    if (delta < 0.0) {
                        uvalue = vdataset.getXValue(series, item);
                        lvalue = uvalue + delta;
                    }
                    else {
                        lvalue = vdataset.getXValue(series, item);
                    }
                    minimum = Math.min(minimum, lvalue);
                    item++;
                }
                series++;
            }
        }
        else {
            int series = 0;
            while (series < seriesCount) {
                int itemCount = dataset.getItemCount(series);
                int item = 0;
                while (item < itemCount){
                    lvalue = dataset.getXValue(series, item);
                    minimum = Math.min(minimum, lvalue);
                    item++;
                }
                series++;
            }
        }
        return minimum;
    }
```

Step 4:

Now we take the Co-Slice code and perform another Sliding on {maximum}.



```
1          double maximum = Double.NEGATIVE_INFINITY;
2          int seriesCount = dataset.getSeriesCount();
3          double lvalue;
4          double uvalue;
5          if (dataset instanceof VectorXYDataset) {
6              VectorXYDataset vdataset = (VectorXYDataset) dataset;
7              int series = 0;
8              while (series < seriesCount){
9                  int itemCount = dataset.getItemCount(series);
10                 int item=0;
11                 while (item < itemCount){
12                     double delta = vdataset.getVectorXValue(series, item);
13                     if (delta < 0.0) {
14                         uvalue = vdataset.getXValue(series, item);
15                     }
16                     else {
17                         lvalue = vdataset.getXValue(series, item);
18                         uvalue = lvalue + delta;
19                     }
20                     maximum = Math.max(maximum, uvalue);
21                     item++;
22                 }
23                 series++;
24             }
25         }
26         else {
27             int series = 0;
28             while (series < seriesCount) {
29                 int itemCount = dataset.getItemCount(series);
30                 int item = 0;
31                 while (item < itemCount){
32                     lvalue = dataset.getXValue(series, item);
33                     uvalue = lvalue;
34                     maximum = Math.max(maximum, uvalue);
35                     item++;
36                 }
37                 series++;
38             }
39         }
```

After we apply Sliding on {maximum} we get:

Slice(exit) = {entry, 1, 2, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 17, 18, 20, 21, 23, 27, 28,29, 30, 31, 32, 33, 34, 35, 37, exit}

Co-Slice(exit) = {}

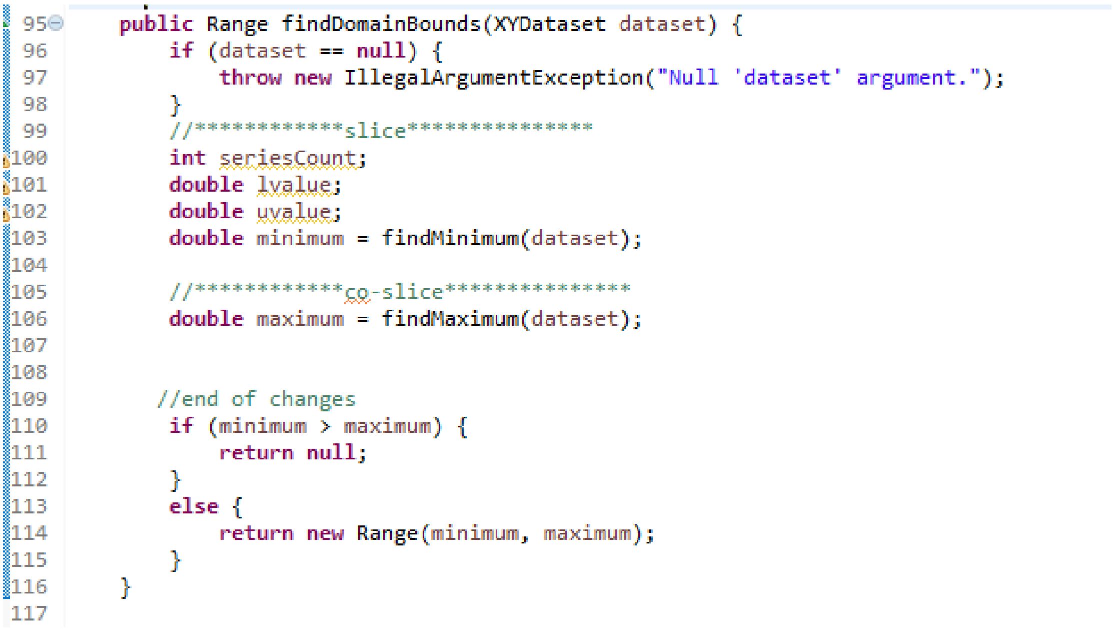
pen1 = {}

pen2 = {}

pen3 = {}.

Step 5:

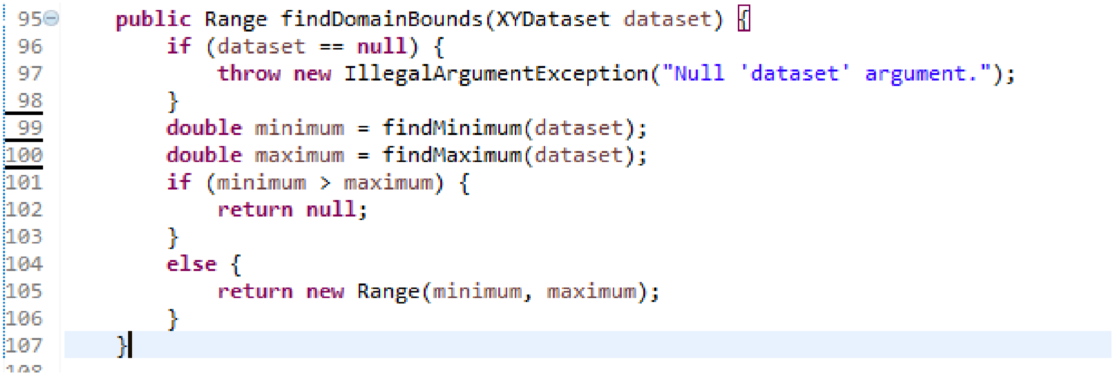
Extract the Slice into a new method.



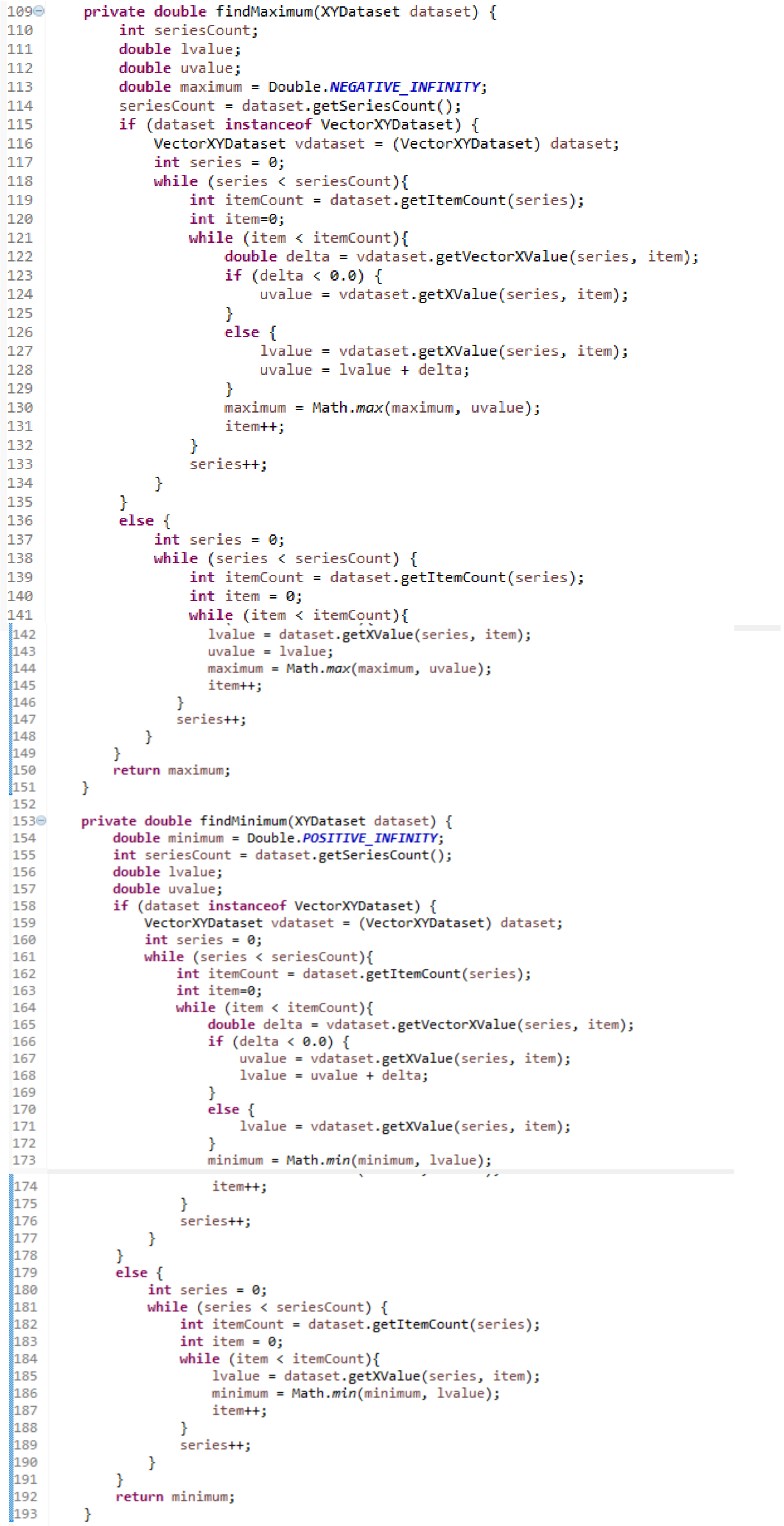
```
 95    public Range findDomainBounds(XYDataset dataset) {
 96        if (dataset == null) {
 97            throw new IllegalArgumentException("Null 'dataset' argument.");
 98        }
 99        //************slice**************
100        int seriesCount;
101        double lvalue;
102        double uvalue;
103        double minimum = findMinimum(dataset);
104
105        //************co-slice**************
106        double maximum = findMaximum(dataset);
107
108
109       //end of changes
110        if (minimum > maximum) {
111            return null;
112        }
113        else {
114            return new Range(minimum, maximum);
115        }
116    }
117
```

Step 6:

dead code elimiation for lines 100-103 ,since there are declaritions for variables that already exists in the extracted methods and now they unnecessary.

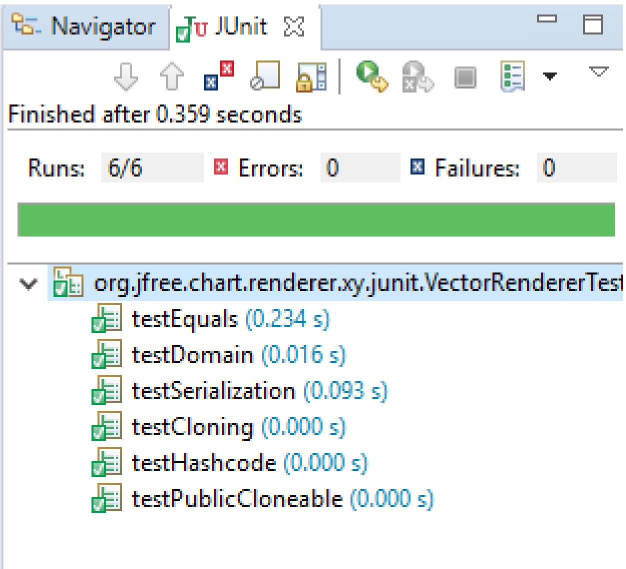


```
 95    public Range findDomainBounds(XYDataset dataset) {
 96        if (dataset == null) {
 97            throw new IllegalArgumentException("Null 'dataset' argument.");
 98        }
 99        double minimum = findMinimum(dataset);
100        double maximum = findMaximum(dataset);
101        if (minimum > maximum) {
102            return null;
103        }
104        else {
105            return new Range(minimum, maximum);
106        }
107    }
108
```



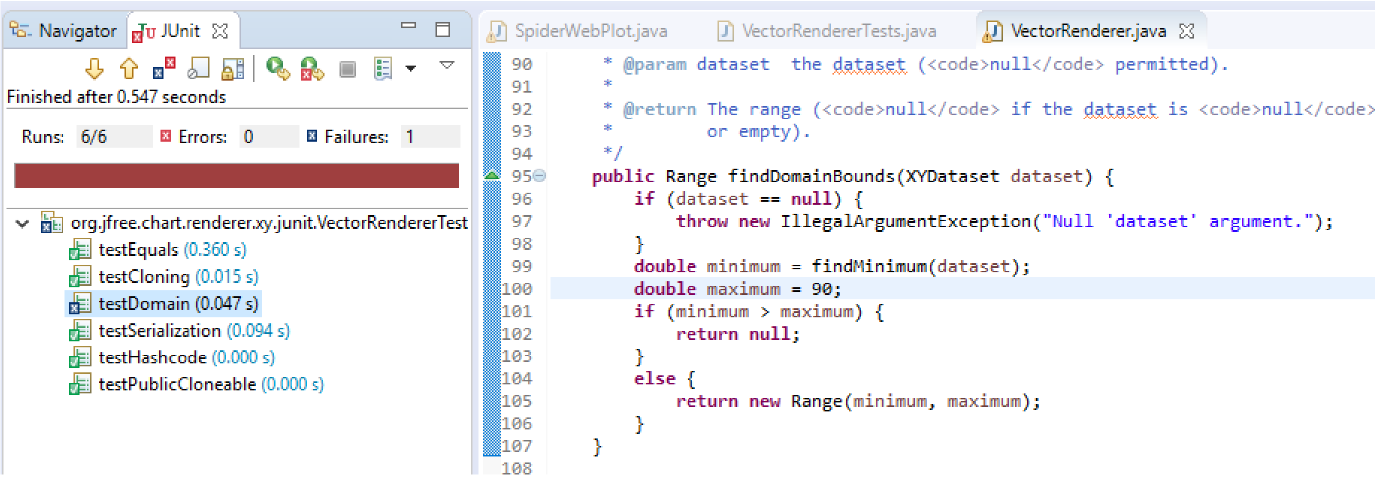
```java
    private double findMaximum(XYDataset dataset) {
        int seriesCount;
        double lvalue;
        double uvalue;
        double maximum = Double.NEGATIVE_INFINITY;
        seriesCount = dataset.getSeriesCount();
        if (dataset instanceof VectorXYDataset) {
            VectorXYDataset vdataset = (VectorXYDataset) dataset;
            int series = 0;
            while (series < seriesCount){
                int itemCount = dataset.getItemCount(series);
                int item=0;
                while (item < itemCount){
                    double delta = vdataset.getVectorXValue(series, item);
                    if (delta < 0.0) {
                        uvalue = vdataset.getXValue(series, item);
                    }
                    else {
                        lvalue = vdataset.getXValue(series, item);
                        uvalue = lvalue + delta;
                    }
                    maximum = Math.max(maximum, uvalue);
                    item++;
                }
                series++;
            }
        }
        else {
            int series = 0;
            while (series < seriesCount) {
                int itemCount = dataset.getItemCount(series);
                int item = 0;
                while (item < itemCount){
                    lvalue = dataset.getXValue(series, item);
                    uvalue = lvalue;
                    maximum = Math.max(maximum, uvalue);
                    item++;
                }
                series++;
            }
        }
        return maximum;
    }

    private double findMinimum(XYDataset dataset) {
        double minimum = Double.POSITIVE_INFINITY;
        int seriesCount = dataset.getSeriesCount();
        double lvalue;
        double uvalue;
        if (dataset instanceof VectorXYDataset) {
            VectorXYDataset vdataset = (VectorXYDataset) dataset;
            int series = 0;
            while (series < seriesCount){
                int itemCount = dataset.getItemCount(series);
                int item=0;
                while (item < itemCount){
                    double delta = vdataset.getVectorXValue(series, item);
                    if (delta < 0.0) {
                        uvalue = vdataset.getXValue(series, item);
                        lvalue = uvalue + delta;
                    }
                    else {
                        lvalue = vdataset.getXValue(series, item);
                    }
                    minimum = Math.min(minimum, lvalue);
                    item++;
                }
                series++;
            }
        }
        else {
            int series = 0;
            while (series < seriesCount) {
                int itemCount = dataset.getItemCount(series);
                int item = 0;
                while (item < itemCount){
                    lvalue = dataset.getXValue(series, item);
                    minimum = Math.min(minimum, lvalue);
                    item++;
                }
                series++;
            }
        }
        return minimum;
    }
```

Let's test the code:



```
Finished after 0.359 seconds

Runs: 6/6    Errors: 0    Failures: 0

org.jfree.chart.renderer.xy.junit.VectorRendererTest
    testEquals (0.234 s)
    testDomain (0.016 s)
    testSerialization (0.093 s)
    testCloning (0.000 s)
    testHashcode (0.000 s)
    testPublicCloneable (0.000 s)
```

Let's inject a piece of code that will cause a failure due to intentional mistake.

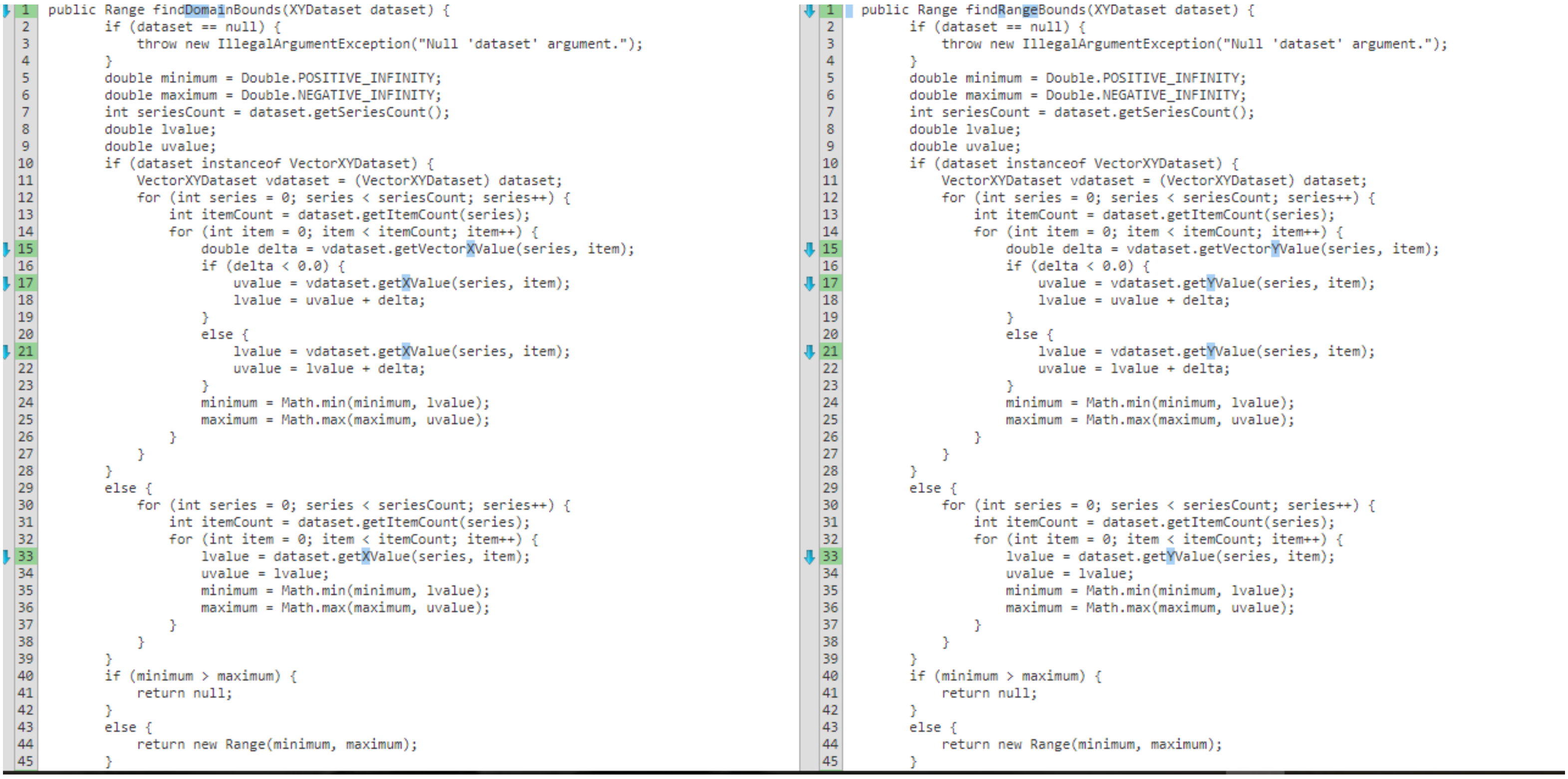


```
Finished after 0.547 seconds

Runs: 6/6    Errors: 0    Failures: 1

org.jfree.chart.renderer.xy.junit.VectorRendererTest
    testEquals (0.360 s)
    testCloning (0.015 s)
    testDomain (0.047 s)
    testSerialization (0.094 s)
    testHashcode (0.000 s)
    testPublicCloneable (0.000 s)
```

```java
     * @param dataset  the dataset (<code>null</code> permitted).
     *
     * @return The range (<code>null</code> if the dataset is <code>null</code>
     *         or empty).
     */
    public Range findDomainBounds(XYDataset dataset) {
        if (dataset == null) {
            throw new IllegalArgumentException("Null 'dataset' argument.");
        }
        double minimum = findMinimum(dataset);
        double maximum = 90;
        if (minimum > maximum) {
            return null;
        }
        else {
            return new Range(minimum, maximum);
        }
    }
```

Step 7:

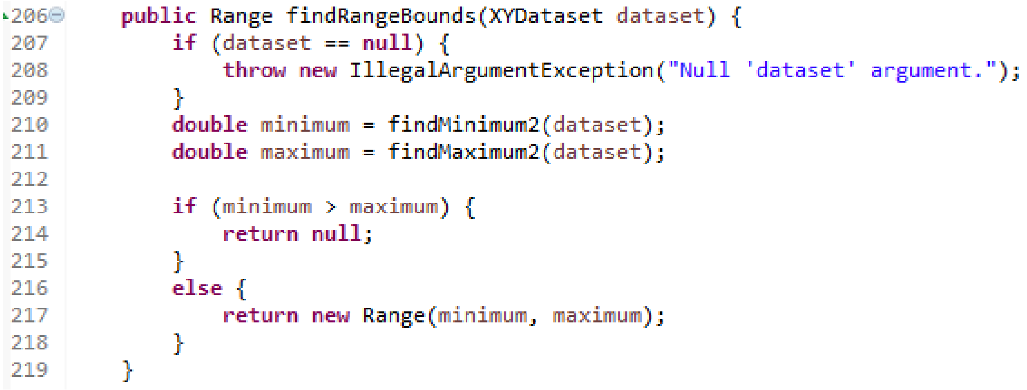
While working on the findDomainBounds method, we noticed that another method called findRangeBounds is almost identical.



```
public Range findDomainBounds(XYDataset dataset) {
        if (dataset == null) {
            throw new IllegalArgumentException("Null 'dataset' argument.");
        }
        double minimum = Double.POSITIVE_INFINITY;
        double maximum = Double.NEGATIVE_INFINITY;
        int seriesCount = dataset.getSeriesCount();
        double lvalue;
        double uvalue;
        if (dataset instanceof VectorXYDataset) {
            VectorXYDataset vdataset = (VectorXYDataset) dataset;
            for (int series = 0; series < seriesCount; series++) {
                int itemCount = dataset.getItemCount(series);
                for (int item = 0; item < itemCount; item++) {
                    double delta = vdataset.getVectorXValue(series, item);
                    if (delta < 0.0) {
                        uvalue = vdataset.getXValue(series, item);
                        lvalue = uvalue + delta;
                    }
                    else {
                        lvalue = vdataset.getXValue(series, item);
                        uvalue = lvalue + delta;
                    }
                    minimum = Math.min(minimum, lvalue);
                    maximum = Math.max(maximum, uvalue);
                }
            }
        }
        else {
            for (int series = 0; series < seriesCount; series++) {
                int itemCount = dataset.getItemCount(series);
                for (int item = 0; item < itemCount; item++) {
                    lvalue = dataset.getXValue(series, item);
                    uvalue = lvalue;
                    minimum = Math.min(minimum, lvalue);
                    maximum = Math.max(maximum, uvalue);
                }
            }
        }
        if (minimum > maximum) {
            return null;
        }
        else {
            return new Range(minimum, maximum);
        }
```

```
public Range findRangeBounds(XYDataset dataset) {
        if (dataset == null) {
            throw new IllegalArgumentException("Null 'dataset' argument.");
        }
        double minimum = Double.POSITIVE_INFINITY;
        double maximum = Double.NEGATIVE_INFINITY;
        int seriesCount = dataset.getSeriesCount();
        double lvalue;
        double uvalue;
        if (dataset instanceof VectorXYDataset) {
            VectorXYDataset vdataset = (VectorXYDataset) dataset;
            for (int series = 0; series < seriesCount; series++) {
                int itemCount = dataset.getItemCount(series);
                for (int item = 0; item < itemCount; item++) {
                    double delta = vdataset.getVectorYValue(series, item);
                    if (delta < 0.0) {
                        uvalue = vdataset.getYValue(series, item);
                        lvalue = uvalue + delta;
                    }
                    else {
                        lvalue = vdataset.getYValue(series, item);
                        uvalue = lvalue + delta;
                    }
                    minimum = Math.min(minimum, lvalue);
                    maximum = Math.max(maximum, uvalue);
                }
            }
        }
        else {
            for (int series = 0; series < seriesCount; series++) {
                int itemCount = dataset.getItemCount(series);
                for (int item = 0; item < itemCount; item++) {
                    lvalue = dataset.getYValue(series, item);
                    uvalue = lvalue;
                    minimum = Math.min(minimum, lvalue);
                    maximum = Math.max(maximum, uvalue);
                }
            }
        }
        if (minimum > maximum) {
            return null;
        }
        else {
            return new Range(minimum, maximum);
        }
```

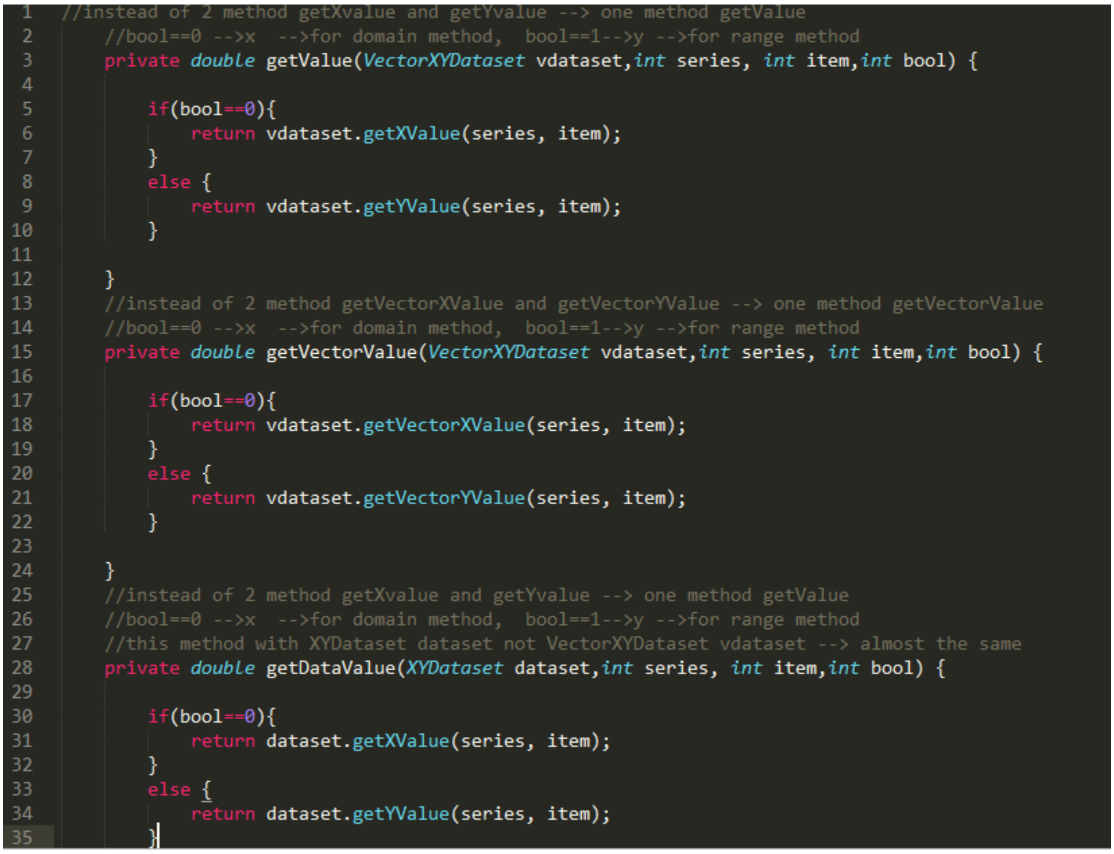
if we do another Sliding to findRangeBound method, we will get the following code:



```
    public Range findRangeBounds(XYDataset dataset) {
        if (dataset == null) {
            throw new IllegalArgumentException("Null 'dataset' argument.");
        }
        double minimum = findMinimum2(dataset);
        double maximum = findMaximum2(dataset);

        if (minimum > maximum) {
            return null;
        }
        else {
            return new Range(minimum, maximum);
        }
    }
```

*findMinimum2 and findMaximum2 are almost identical to findMinimum and findMaximum, the difference is that findMinimum and findMaximum uses `getVectorXValue` and `getXValue` while the findMinimum2 and findMaximum2 uses `getVectorYValue` and `getYValue`.

Our suggestion is to create another three helper methods as follows:



```java
//instead of 2 method getXvalue and getYvalue --> one method getValue
    //bool==0 -->x  -->for domain method,  bool==1-->y -->for range method
    private double getValue(VectorXYDataset vdataset,int series, int item,int bool) {

        if(bool==0){
            return vdataset.getXValue(series, item);
        }
        else {
            return vdataset.getYValue(series, item);
        }

    }
    //instead of 2 method getVectorXValue and getVectorYValue --> one method getVectorValue
    //bool==0 -->x  -->for domain method,  bool==1-->y -->for range method
    private double getVectorValue(VectorXYDataset vdataset,int series, int item,int bool) {

        if(bool==0){
            return vdataset.getVectorXValue(series, item);
        }
        else {
            return vdataset.getVectorYValue(series, item);
        }

    }
    //instead of 2 method getXvalue and getYvalue --> one method getValue
    //bool==0 -->x  -->for domain method,  bool==1-->y -->for range method
    //this method with XYDataset dataset not VectorXYDataset vdataset --> almost the same
    private double getDataValue(XYDataset dataset,int series, int item,int bool) {

        if(bool==0){
            return dataset.getXValue(series, item);
        }
        else {
            return dataset.getYValue(series, item);
        }
```

now we can merge findMinimum and findMinimum2 to one method, and same for findMaximum and findMaximum2.

Instead calling `getVectorXValue` and `getXValue` and `getVectorYValue` and `getYValue` we can call to the new methods.

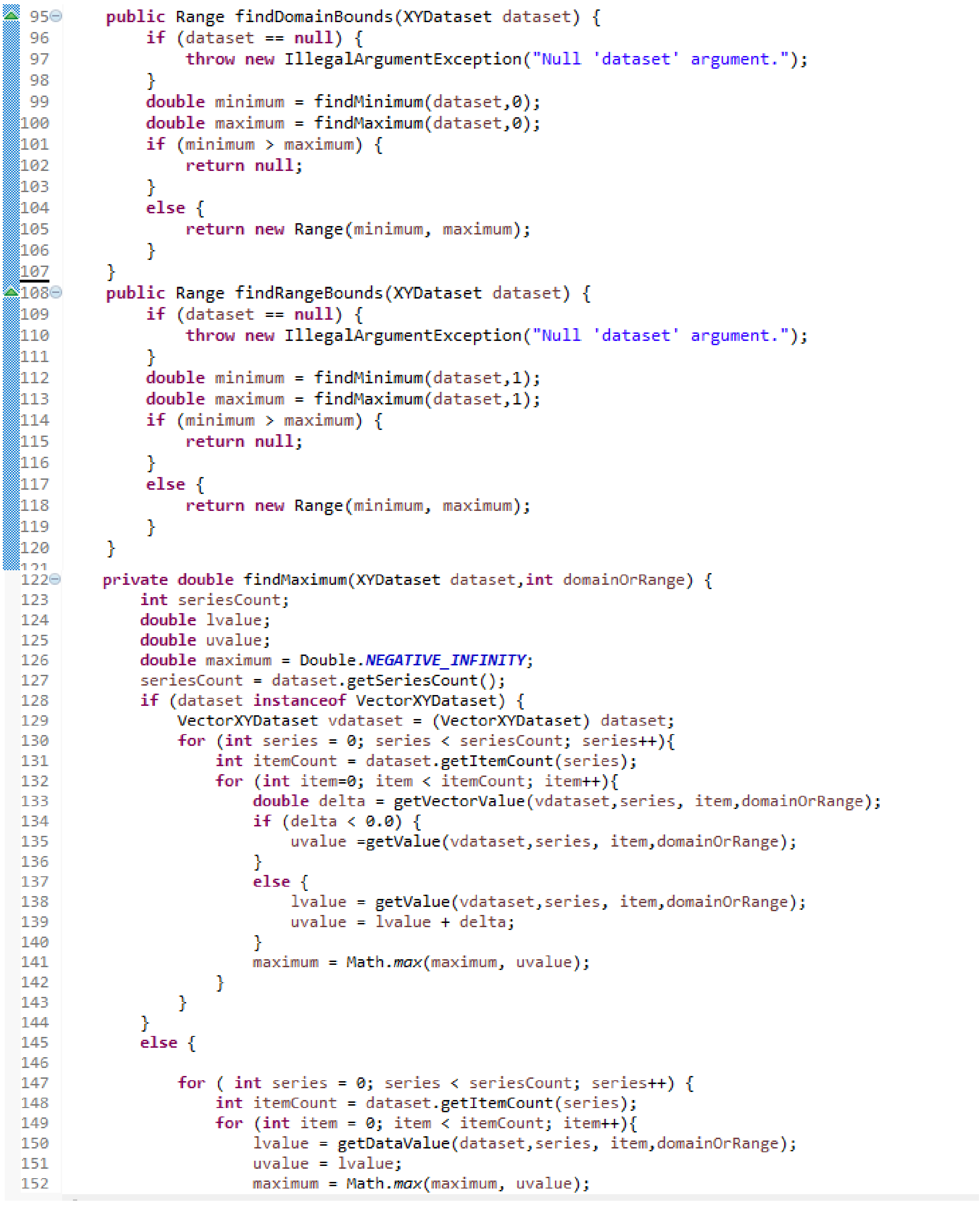
In the findDomainBound we will send another argument (0) that represent for the new methods to use `getVectorXValue` and `getXValue`.

In the findRangeBound we will send another argument (1) that represent for the new methods to use `getVectorYValue` and `getYValue`.

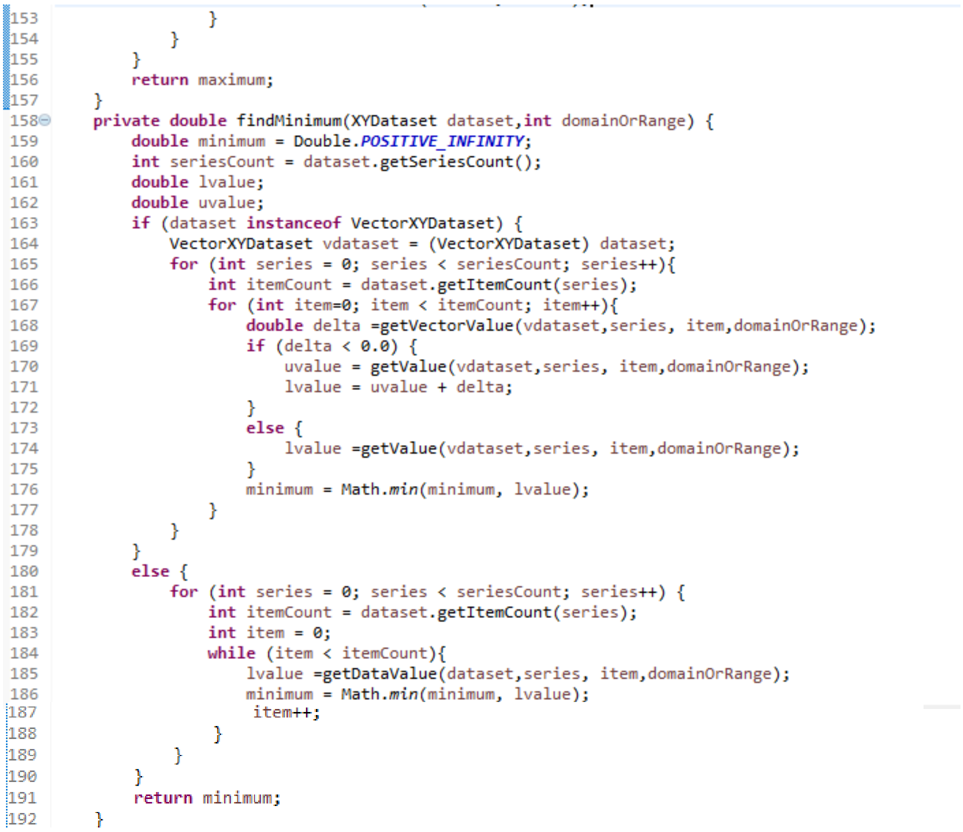
*if we want we can merge findRangeBound and findDomainBound but for that we need to change all the calling to those method (because another argument needed).

The final version will be:

*we return the loops to the previous conditions, while back to for.

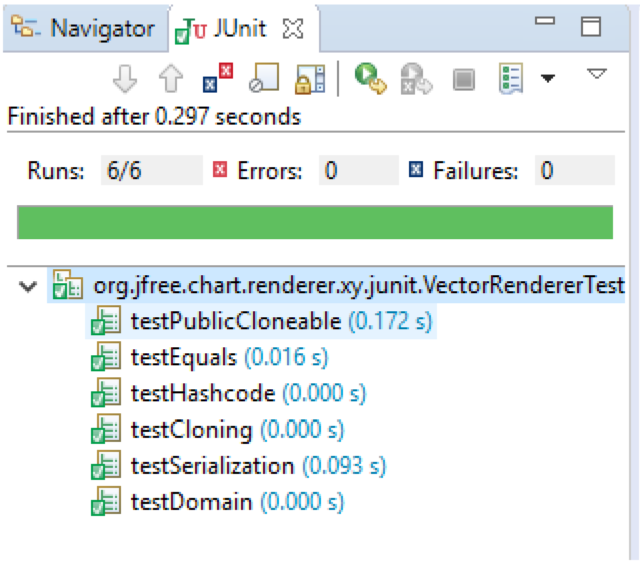


```
    public Range findDomainBounds(XYDataset dataset) {
        if (dataset == null) {
            throw new IllegalArgumentException("Null 'dataset' argument.");
        }
        double minimum = findMinimum(dataset,0);
        double maximum = findMaximum(dataset,0);
        if (minimum > maximum) {
            return null;
        }
        else {
            return new Range(minimum, maximum);
        }
    }
    public Range findRangeBounds(XYDataset dataset) {
        if (dataset == null) {
            throw new IllegalArgumentException("Null 'dataset' argument.");
        }
        double minimum = findMinimum(dataset,1);
        double maximum = findMaximum(dataset,1);
        if (minimum > maximum) {
            return null;
        }
        else {
            return new Range(minimum, maximum);
        }
    }

    private double findMaximum(XYDataset dataset,int domainOrRange) {
        int seriesCount;
        double lvalue;
        double uvalue;
        double maximum = Double.NEGATIVE_INFINITY;
        seriesCount = dataset.getSeriesCount();
        if (dataset instanceof VectorXYDataset) {
            VectorXYDataset vdataset = (VectorXYDataset) dataset;
            for (int series = 0; series < seriesCount; series++){
                int itemCount = dataset.getItemCount(series);
                for (int item=0; item < itemCount; item++){
                    double delta = getVectorValue(vdataset,series, item,domainOrRange);
                    if (delta < 0.0) {
                        uvalue =getValue(vdataset,series, item,domainOrRange);
                    }
                    else {
                        lvalue = getValue(vdataset,series, item,domainOrRange);
                        uvalue = lvalue + delta;
                    }
                    maximum = Math.max(maximum, uvalue);
                }
            }
        }
        else {

            for ( int series = 0; series < seriesCount; series++) {
                int itemCount = dataset.getItemCount(series);
                for (int item = 0; item < itemCount; item++){
                    lvalue = getDataValue(dataset,series, item,domainOrRange);
                    uvalue = lvalue;
                    maximum = Math.max(maximum, uvalue);
```



```
                }
            }
        }
        return maximum;
    }
    private double findMinimum(XYDataset dataset,int domainOrRange) {
        double minimum = Double.POSITIVE_INFINITY;
        int seriesCount = dataset.getSeriesCount();
        double lvalue;
        double uvalue;
        if (dataset instanceof VectorXYDataset) {
            VectorXYDataset vdataset = (VectorXYDataset) dataset;
            for (int series = 0; series < seriesCount; series++){
                int itemCount = dataset.getItemCount(series);
                for (int item=0; item < itemCount; item++){
                    double delta =getVectorValue(vdataset,series, item,domainOrRange);
                    if (delta < 0.0) {
                        uvalue = getValue(vdataset,series, item,domainOrRange);
                        lvalue = uvalue + delta;
                    }
                    else {
                        lvalue =getValue(vdataset,series, item,domainOrRange);
                    }
                    minimum = Math.min(minimum, lvalue);
                }
            }
        }
        else {
            for (int series = 0; series < seriesCount; series++) {
                int itemCount = dataset.getItemCount(series);
                int item = 0;
                while (item < itemCount){
                    lvalue =getDataValue(dataset,series, item,domainOrRange);
                    minimum = Math.min(minimum, lvalue);
                     item++;
                }
            }
        }
        return minimum;
    }
```

Let's test the code:



Finished after 0.297 seconds

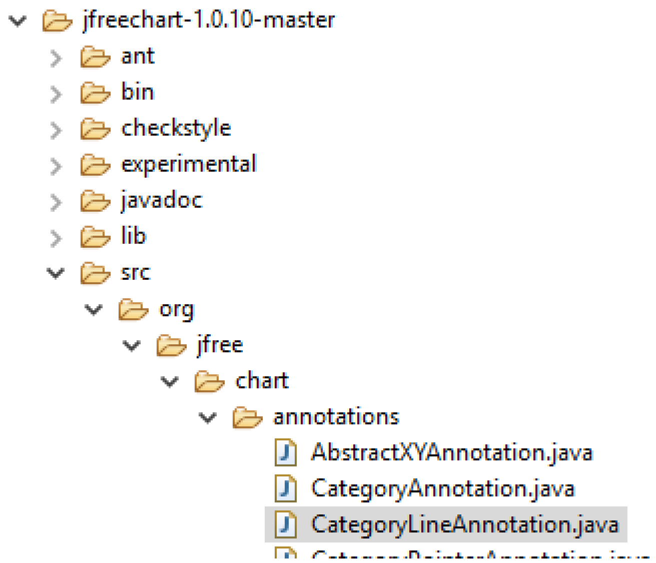
Runs: 6/6   Errors: 0   Failures: 0

```
org.jfree.chart.renderer.xy.junit.VectorRendererTest
    testPublicCloneable (0.172 s)
    testEquals (0.016 s)
    testHashcode (0.000 s)
    testCloning (0.000 s)
    testSerialization (0.093 s)
    testDomain (0.000 s)
```

Bucketing:

We chose a method called "draw" from an open source project which can be found at

https://github.com/alwaqfi/jfreechart-1.0.10/blob/master/src/org/jfree/chart/annotations/CategoryLineAnnotation.java



```
jfreechart-1.0.10-master
  ant
  bin
  checkstyle
  experimental
  javadoc
  lib
  src
    org
      jfree
        chart
          annotations
            AbstractXYAnnotation.java
            CategoryAnnotation.java
            CategoryLineAnnotation.java
            CategoryPointerAnnotation.java
```

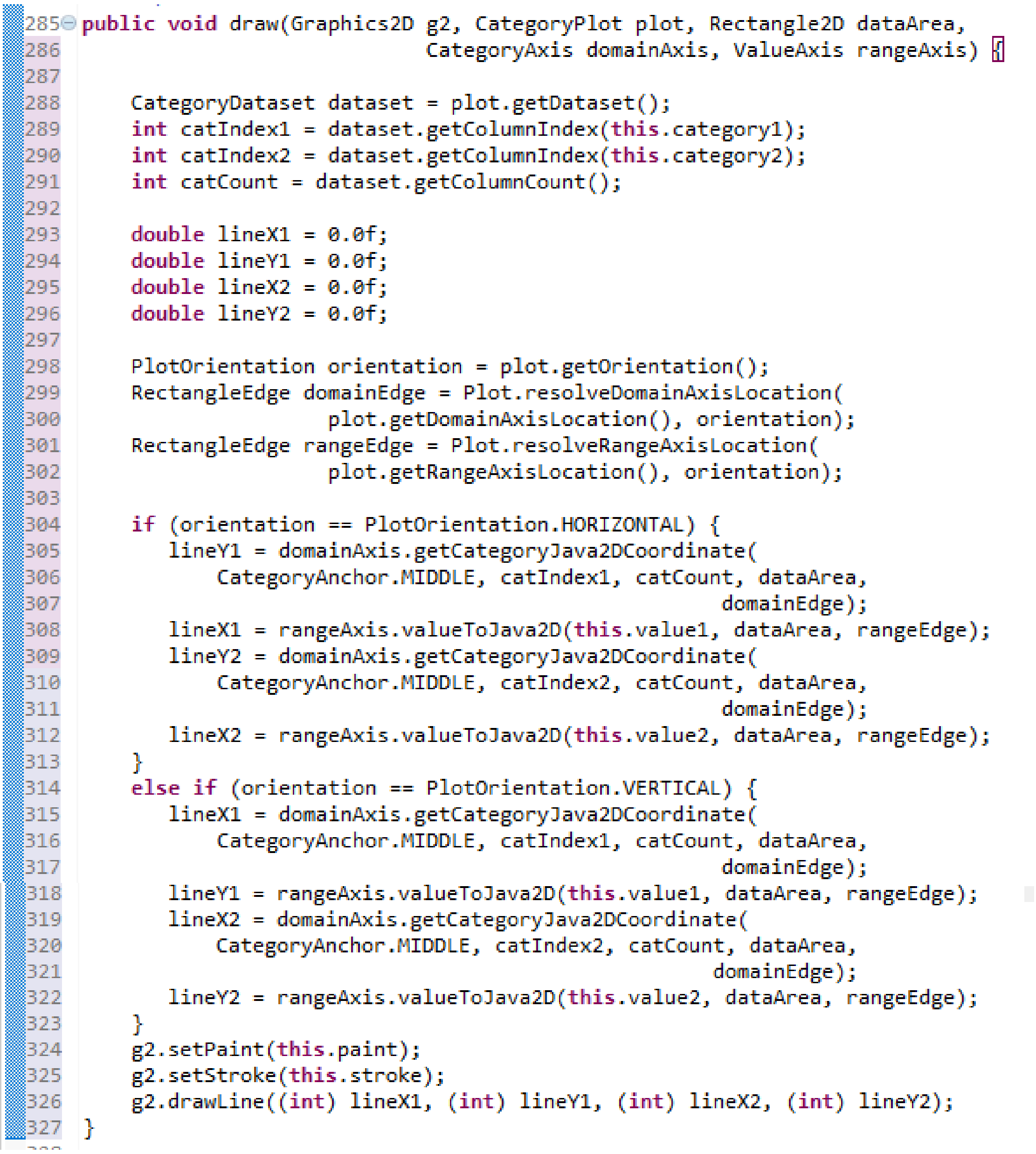
Since the method is too large, we focused on a specific block of code inside it, and applied the bucketing algorithm on that specific block of code.

We noticed that there are definitions of many variables {lineX1, lineX2, lineY1, lineY2} at a time.

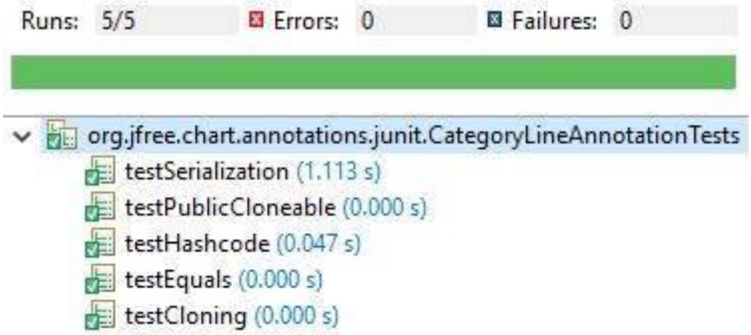
We want the method to be easier to read and maintain by extracting their computations to external methods.

Another reason to perform this extraction is the repeatedly computations for these variables in another methods in this project, by doing so, we can reuse this external methods and avoid repeated code (clone elimination).

Original code:

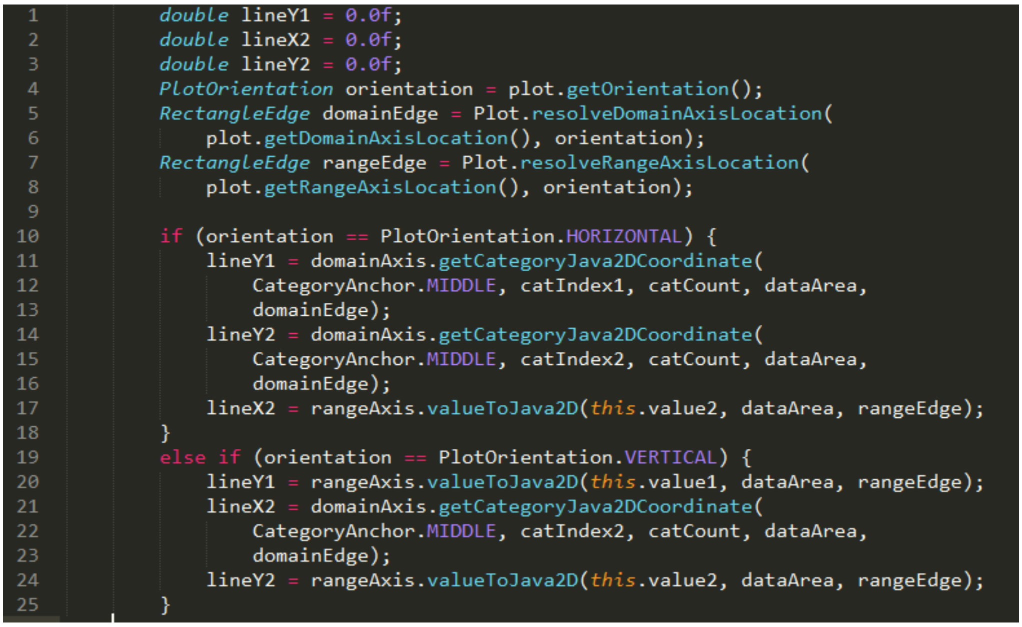


```
public void draw(Graphics2D g2, CategoryPlot plot, Rectangle2D dataArea,
                            CategoryAxis domainAxis, ValueAxis rangeAxis) {

    CategoryDataset dataset = plot.getDataset();
    int catIndex1 = dataset.getColumnIndex(this.category1);
    int catIndex2 = dataset.getColumnIndex(this.category2);
    int catCount = dataset.getColumnCount();

    double lineX1 = 0.0f;
    double lineY1 = 0.0f;
    double lineX2 = 0.0f;
    double lineY2 = 0.0f;

    PlotOrientation orientation = plot.getOrientation();
    RectangleEdge domainEdge = Plot.resolveDomainAxisLocation(
                    plot.getDomainAxisLocation(), orientation);
    RectangleEdge rangeEdge = Plot.resolveRangeAxisLocation(
                    plot.getRangeAxisLocation(), orientation);

    if (orientation == PlotOrientation.HORIZONTAL) {
       lineY1 = domainAxis.getCategoryJava2DCoordinate(
           CategoryAnchor.MIDDLE, catIndex1, catCount, dataArea,
                                                    domainEdge);
       lineX1 = rangeAxis.valueToJava2D(this.value1, dataArea, rangeEdge);
       lineY2 = domainAxis.getCategoryJava2DCoordinate(
           CategoryAnchor.MIDDLE, catIndex2, catCount, dataArea,
                                                    domainEdge);
       lineX2 = rangeAxis.valueToJava2D(this.value2, dataArea, rangeEdge);
    }
    else if (orientation == PlotOrientation.VERTICAL) {
       lineX1 = domainAxis.getCategoryJava2DCoordinate(
           CategoryAnchor.MIDDLE, catIndex1, catCount, dataArea,
                                                    domainEdge);
       lineY1 = rangeAxis.valueToJava2D(this.value1, dataArea, rangeEdge);
       lineX2 = domainAxis.getCategoryJava2DCoordinate(
           CategoryAnchor.MIDDLE, catIndex2, catCount, dataArea,
                                                   domainEdge);
       lineY2 = rangeAxis.valueToJava2D(this.value2, dataArea, rangeEdge);
    }
    g2.setPaint(this.paint);
    g2.setStroke(this.stroke);
    g2.drawLine((int) lineX1, (int) lineY1, (int) lineX2, (int) lineY2);
}
```



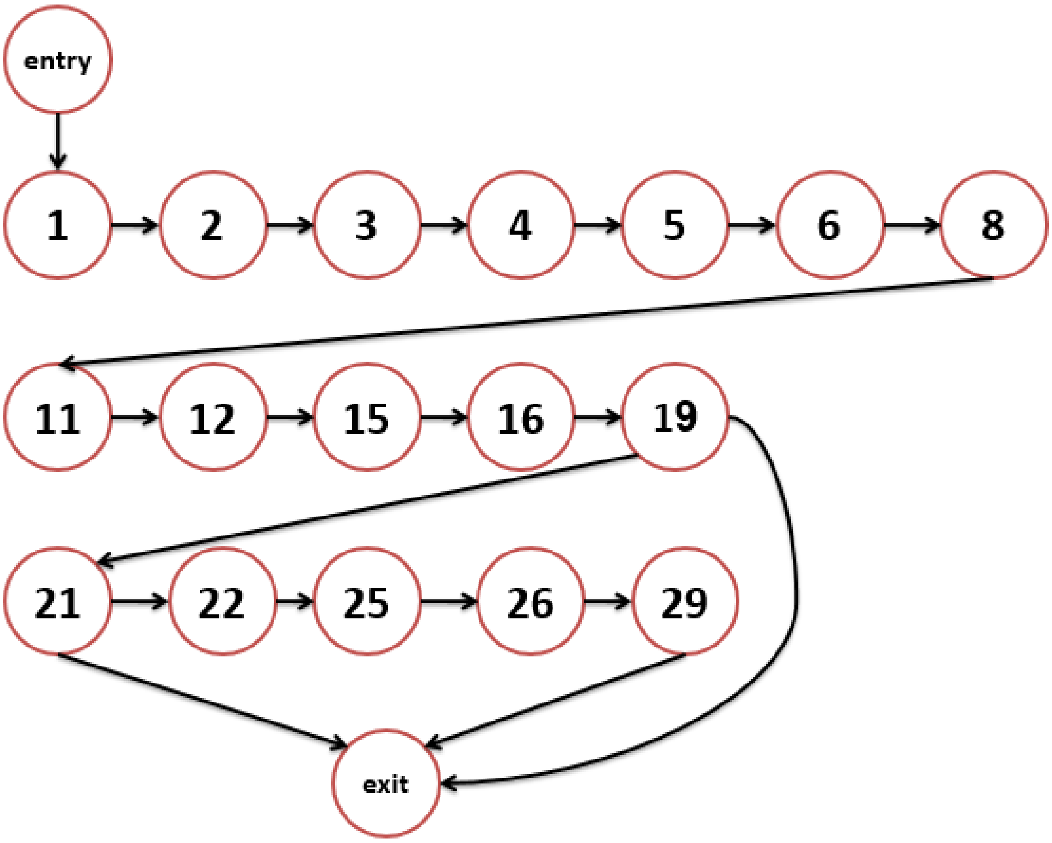
```
Runs: 5/5        Errors: 0        Failures: 0

org.jfree.chart.annotations.junit.CategoryLineAnnotationTests
    testSerialization (1.113 s)
    testPublicCloneable (0.000 s)
    testHashcode (0.047 s)
    testEquals (0.000 s)
    testCloning (0.000 s)
```

We chose the block of code from lines 293-323, for convenience we numbered those lines from 1-25 .



```
1          double lineY1 = 0.0f;
2          double lineX2 = 0.0f;
3          double lineY2 = 0.0f;
4          PlotOrientation orientation = plot.getOrientation();
5          RectangleEdge domainEdge = Plot.resolveDomainAxisLocation(
6              plot.getDomainAxisLocation(), orientation);
7          RectangleEdge rangeEdge = Plot.resolveRangeAxisLocation(
8              plot.getRangeAxisLocation(), orientation);
9
10         if (orientation == PlotOrientation.HORIZONTAL) {
11             lineY1 = domainAxis.getCategoryJava2DCoordinate(
12                 CategoryAnchor.MIDDLE, catIndex1, catCount, dataArea,
13                 domainEdge);
14             lineY2 = domainAxis.getCategoryJava2DCoordinate(
15                 CategoryAnchor.MIDDLE, catIndex2, catCount, dataArea,
16                 domainEdge);
17             lineX2 = rangeAxis.valueToJava2D(this.value2, dataArea, rangeEdge);
18         }
19         else if (orientation == PlotOrientation.VERTICAL) {
20             lineY1 = rangeAxis.valueToJava2D(this.value1, dataArea, rangeEdge);
21             lineX2 = domainAxis.getCategoryJava2DCoordinate(
22                 CategoryAnchor.MIDDLE, catIndex2, catCount, dataArea,
23                 domainEdge);
24             lineY2 = rangeAxis.valueToJava2D(this.value2, dataArea, rangeEdge);
25         }
```

Build the CFG:



Build the PDG:

Before we can perform the bucketing, we need to compute the PDG (we did it with {entry, exit} nodes for the correctness of the PDG, although we won't need them for the bucketing).

Edge	type
(Entry,1)	**Control**
(Entry,2)	**Control**
(Entry,3)	**Control**
(Entry,4)	**Control**
(Entry,5)	**Control**
(Entry,6)	**Control**
(Entry,8)	**Control**
(Entry,11)	**Control**
(Entry,21)	**Control**
(11,12)	**Control**
(11,15)	**Control**
(11,16)	**Control**
(11,19)	**Control**
(21,22)	**Control**
(21,25)	**Control**
(21,26)	**Control**
(21,29)	**Control**
(29, exit)	**Control**
(19, exit)	**Control**

Edge	type	Vars
(1,15)	output	**{lineX1}**
(1,22)	output	**{lineX1}**
(2,12)	output	**{lineY1}**
(2,25)	output	**{lineY1}**
(3,19)	output	**{lineX2}**
(3,26)	output	**{lineX2}**
(4,16)	output	**{lineY2}**
(4,29)	output	**{lineY2}**

Edge	type	Vars
(5,6)	Flow	**{ orientation }**
(5,8)	Flow	**{ orientation }**
(5,11)	Flow	**{ orientation }**
(5,21)	Flow	**{ orientation }**
(entry,5)	Flow	**{ plot }**
(entry,6)	Flow	**{ plot }**
(entry,8)	Flow	**{ plot }**
(6,12)	Flow	**{ domainEdge}**
(6,16)	Flow	**{ domainEdge}**
(6,22)	Flow	**{ domainEdge}**
(6,26)	Flow	**{ domainEdge}**
(8,15)	Flow	**{rangeEdge}**
(8,19)	Flow	**{rangeEdge}**
(8,25)	Flow	**{rangeEdge}**
(8,29)	Flow	**{rangeEdge}**
(entry,12)	Flow	**{domainAxis}**
(entry,16)	Flow	**{domainAxis}**
(entry,22)	Flow	**{domainAxis}**
(entry,26)	Flow	**{domainAxis}**
(entry,15)	Flow	**{rangeAxis}**
(entry,19)	Flow	**{rangeAxis}**
(entry,25)	Flow	**{rangeAxis}**
(entry,29)	Flow	**{rangeAxis}**
(entry,12)	Flow	**{catIndex1}**
(entry,22)	Flow	**{catIndex1}**
(entry,16)	Flow	**{catIndex2}**
(entry,26)	Flow	**{catIndex2}**

Edge	type	Vars
(entry,12)	Flow	**{catCount}**
(entry,16)	Flow	**{catCount}**
(entry,22)	Flow	**{catCount}**
(entry,26)	Flow	**{catCount}**
(entry,12)	Flow	**{dataArea}**
(entry,15)	Flow	**{dataArea}**
(entry,16)	Flow	**{dataArea}**
(entry,19)	Flow	**{dataArea}**
(entry,22)	Flow	**{dataArea}**
(entry,25)	Flow	**{dataArea}**
(entry,26)	Flow	**{dataArea}**
(entry,29)	Flow	**{dataArea}**
(entry,22)	Flow	**{this.value1}**
(entry,15)	Flow	**{this.value1}**
(entry,25)	Flow	**{this.value1}**
(entry,29)	Flow	**{this.value2}**
(entry,19)	Flow	**{this.value2}**
(15,exit)	Flow	**{lineX1}**
(22,exit)	Flow	**{lineX1}**
(12,exit)	Flow	**{lineY1}**
(25,exit)	Flow	**{lineY1}**
(19,exit)	Flow	**{lineX2}**
(26,exit)	Flow	**{lineX2}**
(16,exit)	Flow	**{lineY2}**
(29,exit)	Flow	**{lineY2}**

Build the Slide-DG:

For building the Slide-DG, we will remove all the predicate nodes and {entry ,exit} and their edges.

We will copy all the other nodes and their edges to a new graph and add the slide dependence to the graph.

For computing the slide dependence, we first need to compute the slide for each node.

slide(1)= {1}

slide(2)= {2}

slide(3)= {3}

slide(4)= {4}

slide(5)= {5}

slide(12)= {12,11}

slide(16)= {16,11}

slide(22)= {22,21}

slide(26)= {26,21}

slide(6)= {6}

slide(15)= {15,11}

slide(19)= {19,11}

slide(25)= {25,21}

slide(29)= {29,21}

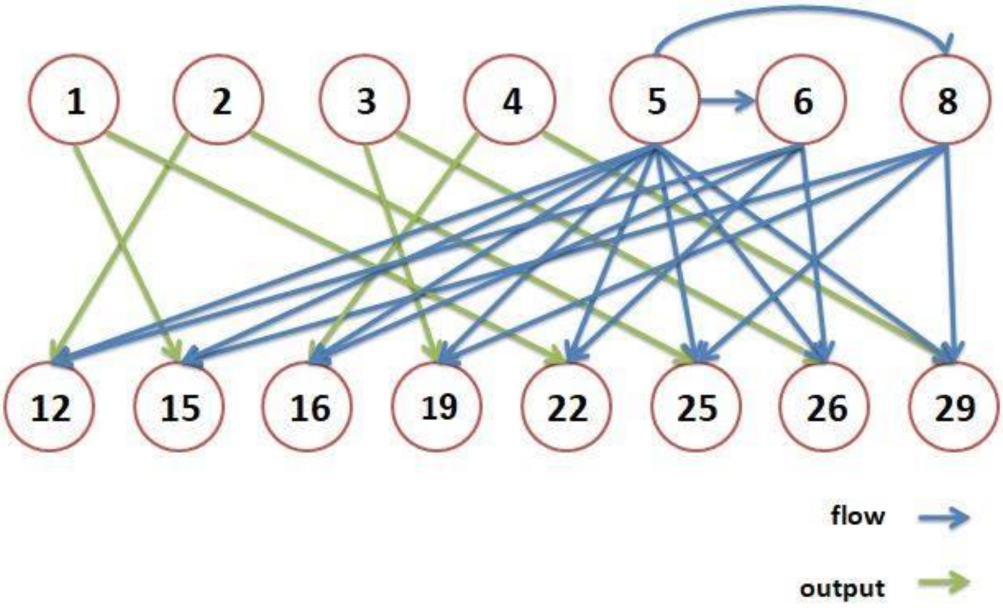
slide(8)= {8}

Now we get this new slide dependence edges:

Edge
(5,12)
(5,19)
(5,16)
(5,15)
(5,22)
(5,26)
(5,29)
(5,25)

After those steps we get this Slide-DG:

Edge
(5,6)
(5,8)
(6,12)
(6,16)
(6,22)
(6,26)
(8,15)
(8,19)
(8,25)
(8,29)
(1,15)
(1,22)
(2,12)
(2,25)
(3,19)
(3,26)
(4,16)
(4,29)
(5,12)
(5,19)
(5,16)
(5,15)
(5,22)
(5,26)
(5,29)
(5,25)



Our purpose is to extract each variable computation to external method, we noticed that the order of the extractions is arbitrary, we chose to apply the bucketing algorithm on "lineX1".

For that purpose we chose M = {1,15,22} , since those lines are the definitions to "lineX1" .

Reaching -M = {1,15,22,5,6,8}

M -Reachable = {1,15,22}

Buckets are computed as follows:

- Before = {Reaching-M \ M-Reachable}

- After = {M-Reachable \ Reaching-M}

- Marked = {Reaching-M ∩ M-Reachable} Predicates corresponding to the nodes of each bucket are added as well.

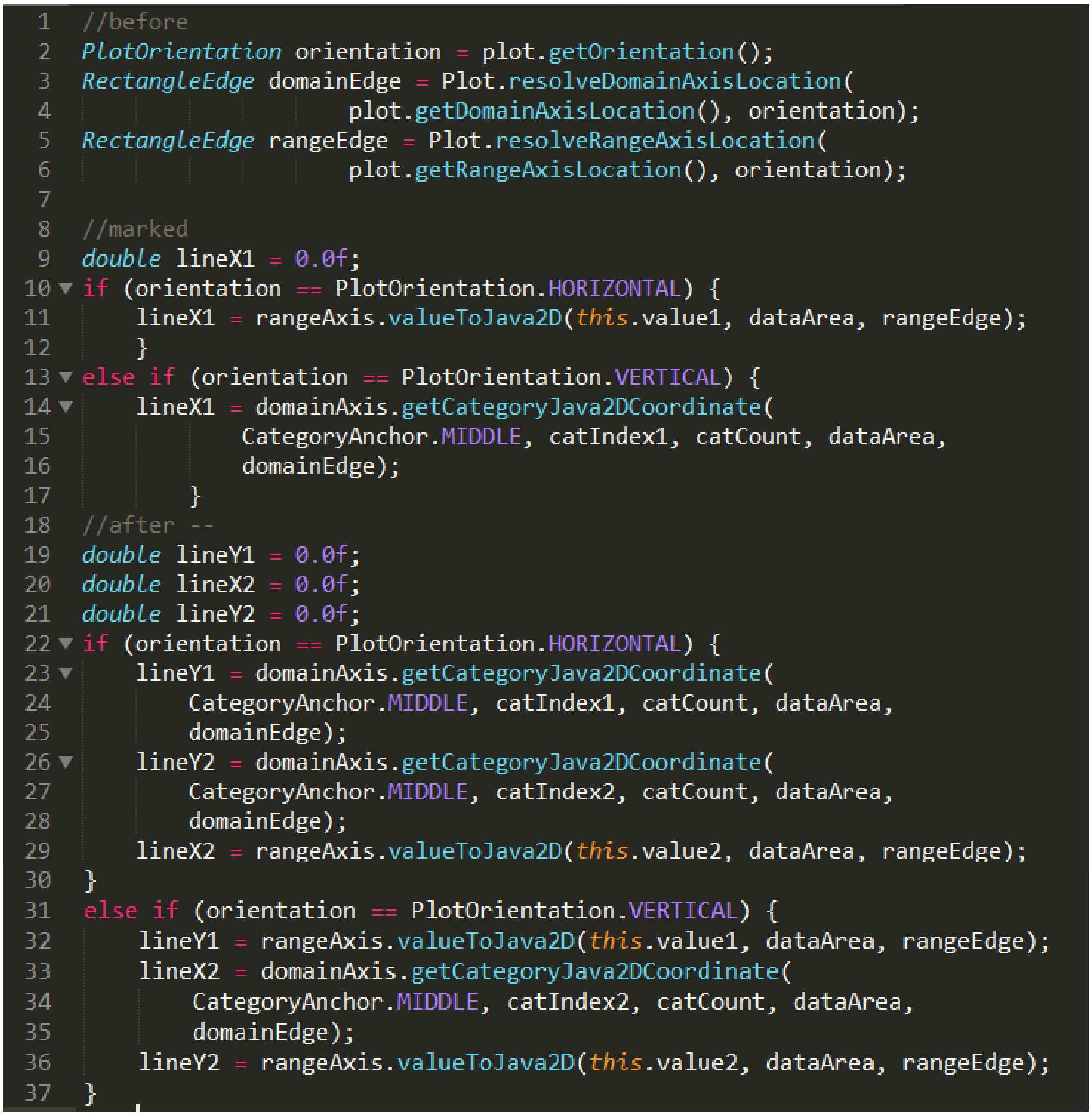
Therefore:

Before = {6,5,8}

After = {}

Marked = {1,15,22}

Result after first step:



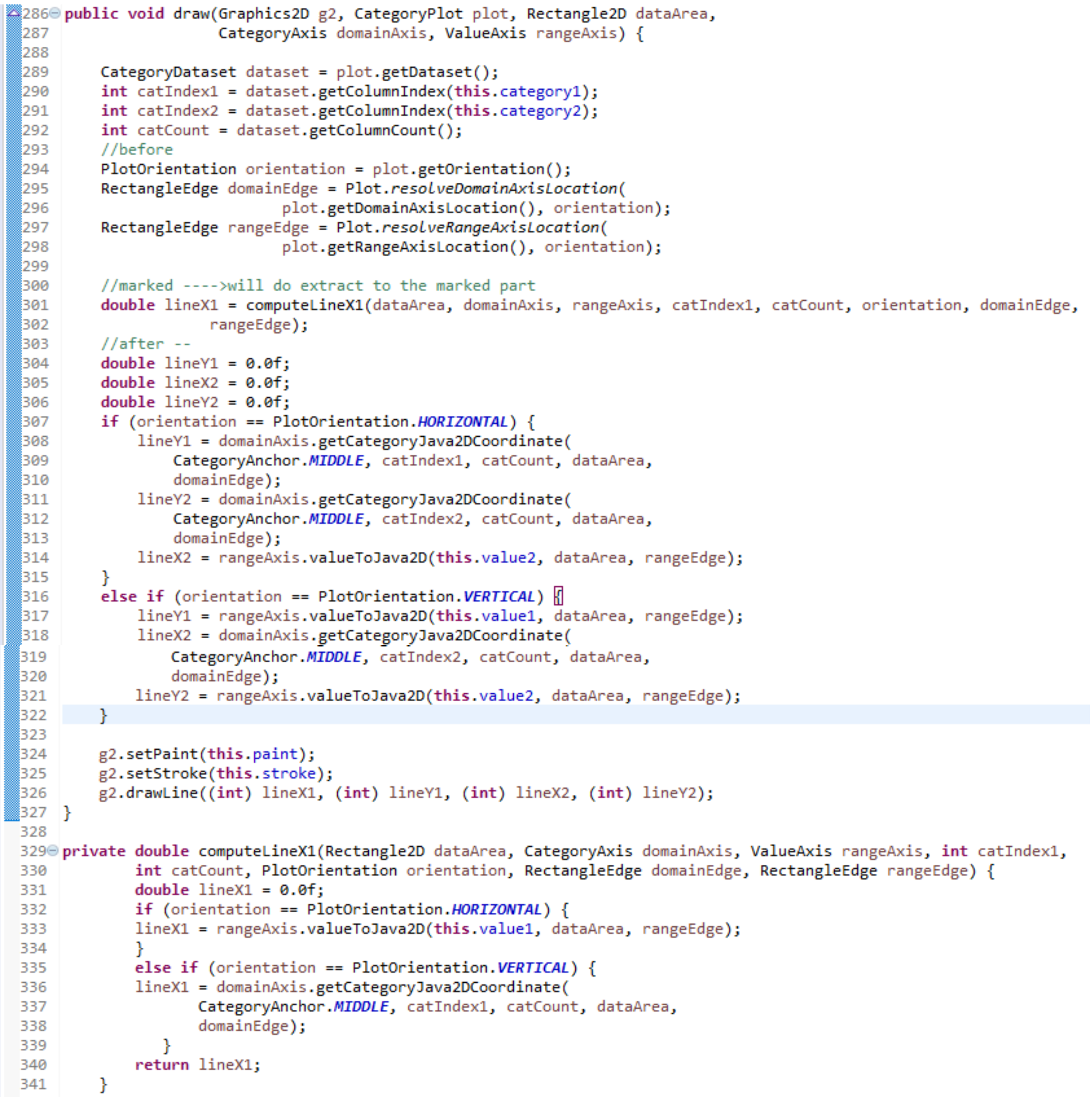
```
1   //before
2   PlotOrientation orientation = plot.getOrientation();
3   RectangleEdge domainEdge = Plot.resolveDomainAxisLocation(
4                       plot.getDomainAxisLocation(), orientation);
5   RectangleEdge rangeEdge = Plot.resolveRangeAxisLocation(
6                       plot.getRangeAxisLocation(), orientation);
7
8   //marked
9   double lineX1 = 0.0f;
10  if (orientation == PlotOrientation.HORIZONTAL) {
11      lineX1 = rangeAxis.valueToJava2D(this.value1, dataArea, rangeEdge);
12      }
13  else if (orientation == PlotOrientation.VERTICAL) {
14      lineX1 = domainAxis.getCategoryJava2DCoordinate(
15              CategoryAnchor.MIDDLE, catIndex1, catCount, dataArea,
16              domainEdge);
17          }
18  //after --
19  double lineY1 = 0.0f;
20  double lineX2 = 0.0f;
21  double lineY2 = 0.0f;
22  if (orientation == PlotOrientation.HORIZONTAL) {
23      lineY1 = domainAxis.getCategoryJava2DCoordinate(
24          CategoryAnchor.MIDDLE, catIndex1, catCount, dataArea,
25          domainEdge);
26      lineY2 = domainAxis.getCategoryJava2DCoordinate(
27          CategoryAnchor.MIDDLE, catIndex2, catCount, dataArea,
28          domainEdge);
29      lineX2 = rangeAxis.valueToJava2D(this.value2, dataArea, rangeEdge);
30  }
31  else if (orientation == PlotOrientation.VERTICAL) {
32      lineY1 = rangeAxis.valueToJava2D(this.value1, dataArea, rangeEdge);
33      lineX2 = domainAxis.getCategoryJava2DCoordinate(
34          CategoryAnchor.MIDDLE, catIndex2, catCount, dataArea,
35          domainEdge);
36      lineY2 = rangeAxis.valueToJava2D(this.value2, dataArea, rangeEdge);
37  }
```

After applying the bucketing, we noticed that lines 2-6 will always appear in the "before" since we need those lines to define those variables.

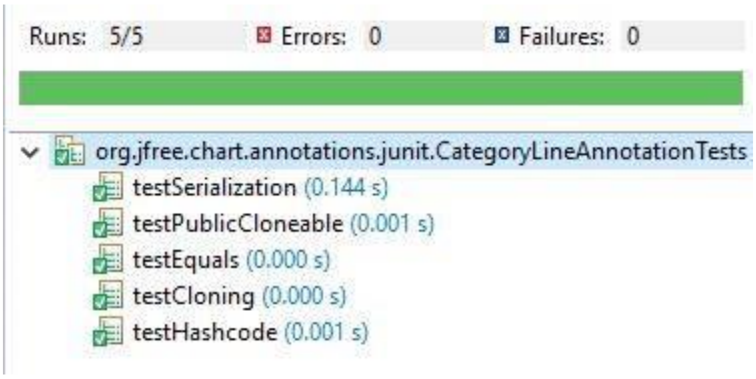
Since lines 19-37 are arbitrary, they will be in the "after" and in the following steps we will apply the bucketing only on those lines.

The marked bucket will be extracted to an external method called computeLineX1.

The result will be:



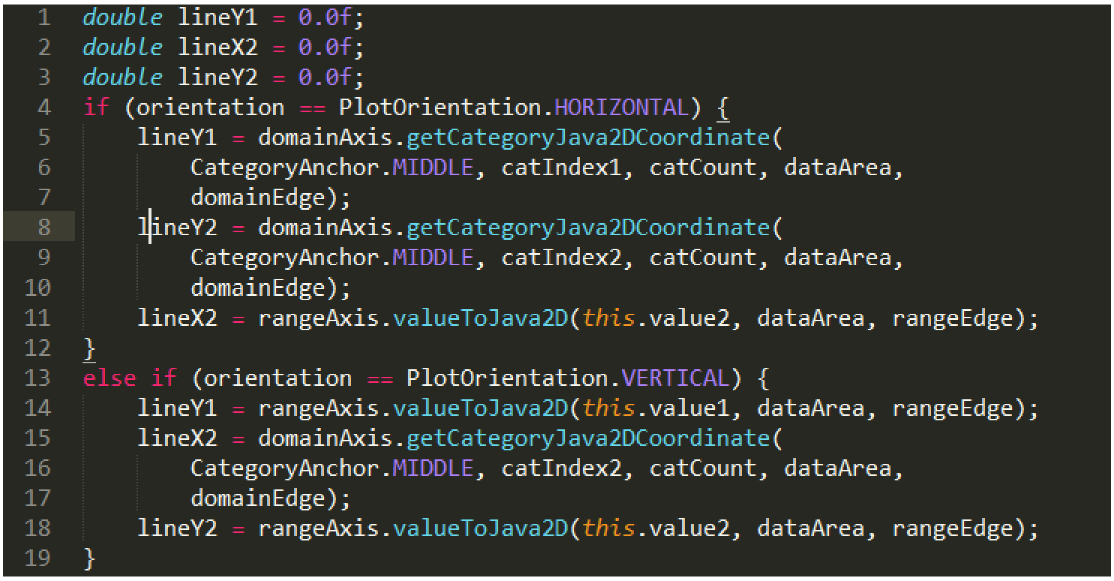
```
public void draw(Graphics2D g2, CategoryPlot plot, Rectangle2D dataArea,
                 CategoryAxis domainAxis, ValueAxis rangeAxis) {

    CategoryDataset dataset = plot.getDataset();
    int catIndex1 = dataset.getColumnIndex(this.category1);
    int catIndex2 = dataset.getColumnIndex(this.category2);
    int catCount = dataset.getColumnCount();
    //before
    PlotOrientation orientation = plot.getOrientation();
    RectangleEdge domainEdge = Plot.resolveDomainAxisLocation(
                        plot.getDomainAxisLocation(), orientation);
    RectangleEdge rangeEdge = Plot.resolveRangeAxisLocation(
                        plot.getRangeAxisLocation(), orientation);

    //marked ---->will do extract to the marked part
    double lineX1 = computeLineX1(dataArea, domainAxis, rangeAxis, catIndex1, catCount, orientation, domainEdge,
                rangeEdge);
    //after --
    double lineY1 = 0.0f;
    double lineX2 = 0.0f;
    double lineY2 = 0.0f;
    if (orientation == PlotOrientation.HORIZONTAL) {
        lineY1 = domainAxis.getCategoryJava2DCoordinate(
            CategoryAnchor.MIDDLE, catIndex1, catCount, dataArea,
            domainEdge);
        lineY2 = domainAxis.getCategoryJava2DCoordinate(
            CategoryAnchor.MIDDLE, catIndex2, catCount, dataArea,
            domainEdge);
        lineX2 = rangeAxis.valueToJava2D(this.value2, dataArea, rangeEdge);
    }
    else if (orientation == PlotOrientation.VERTICAL) {
        lineY1 = rangeAxis.valueToJava2D(this.value1, dataArea, rangeEdge);
        lineX2 = domainAxis.getCategoryJava2DCoordinate(
            CategoryAnchor.MIDDLE, catIndex2, catCount, dataArea,
            domainEdge);
        lineY2 = rangeAxis.valueToJava2D(this.value2, dataArea, rangeEdge);
    }

    g2.setPaint(this.paint);
    g2.setStroke(this.stroke);
    g2.drawLine((int) lineX1, (int) lineY1, (int) lineX2, (int) lineY2);
}

private double computeLineX1(Rectangle2D dataArea, CategoryAxis domainAxis, ValueAxis rangeAxis, int catIndex1,
        int catCount, PlotOrientation orientation, RectangleEdge domainEdge, RectangleEdge rangeEdge) {
        double lineX1 = 0.0f;
        if (orientation == PlotOrientation.HORIZONTAL) {
        lineX1 = rangeAxis.valueToJava2D(this.value1, dataArea, rangeEdge);
        }
        else if (orientation == PlotOrientation.VERTICAL) {
        lineX1 = domainAxis.getCategoryJava2DCoordinate(
               CategoryAnchor.MIDDLE, catIndex1, catCount, dataArea,
               domainEdge);
           }
        return lineX1;
    }
```



```
Runs: 5/5     Errors: 0     Failures: 0

org.jfree.chart.annotations.junit.CategoryLineAnnotationTests
    testSerialization (0.144 s)
    testPublicCloneable (0.001 s)
    testEquals (0.000 s)
    testCloning (0.000 s)
    testHashcode (0.001 s)
```

Step 2:

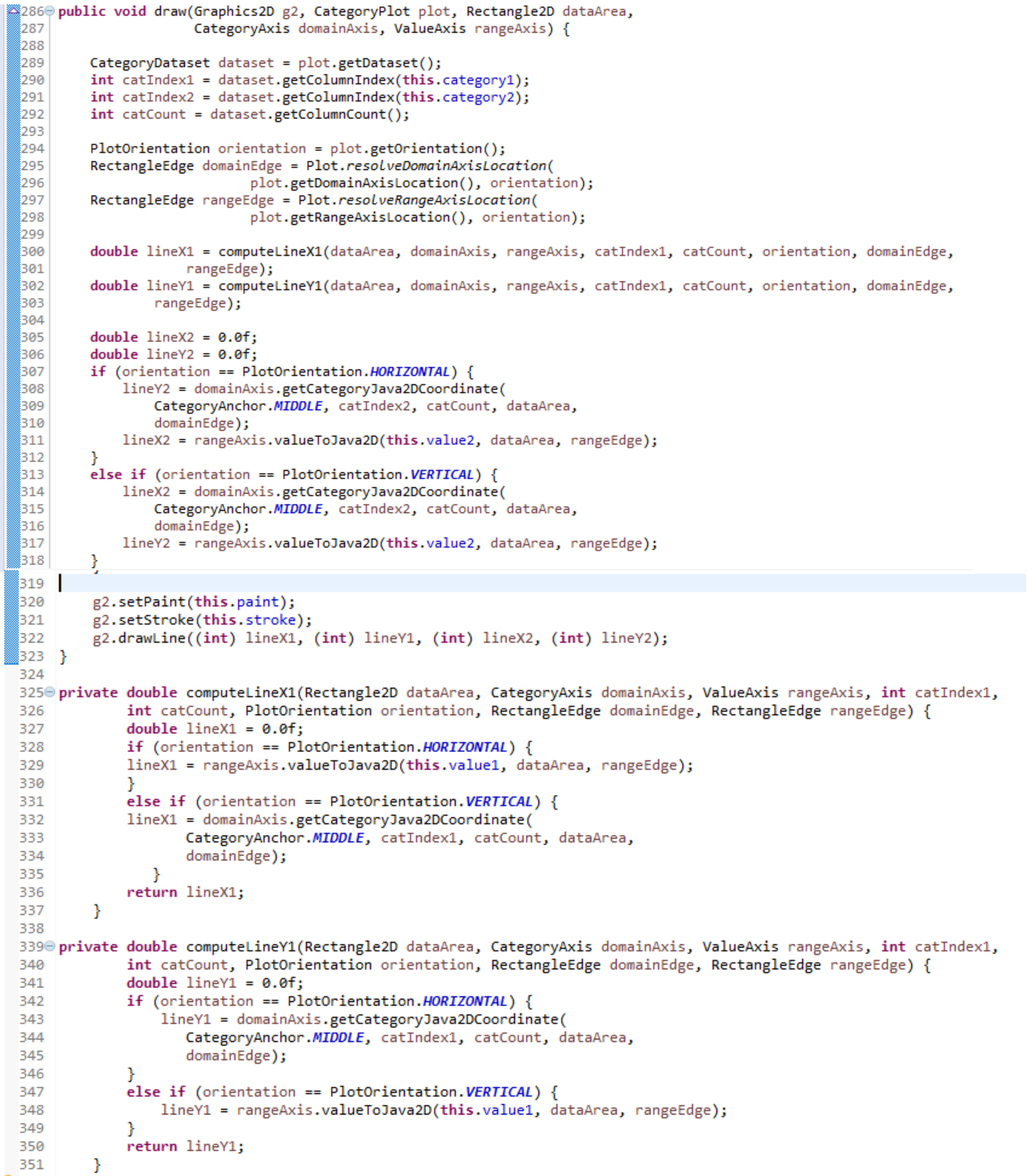
we apply the bucketing algorithm on lineY1.



```
 1  double lineY1 = 0.0f;
 2  double lineX2 = 0.0f;
 3  double lineY2 = 0.0f;
 4  if (orientation == PlotOrientation.HORIZONTAL) {
 5      lineY1 = domainAxis.getCategoryJava2DCoordinate(
 6          CategoryAnchor.MIDDLE, catIndex1, catCount, dataArea,
 7          domainEdge);
 8      lineY2 = domainAxis.getCategoryJava2DCoordinate(
 9          CategoryAnchor.MIDDLE, catIndex2, catCount, dataArea,
10          domainEdge);
11      lineX2 = rangeAxis.valueToJava2D(this.value2, dataArea, rangeEdge);
12  }
13  else if (orientation == PlotOrientation.VERTICAL) {
14      lineY1 = rangeAxis.valueToJava2D(this.value1, dataArea, rangeEdge);
15      lineX2 = domainAxis.getCategoryJava2DCoordinate(
16          CategoryAnchor.MIDDLE, catIndex2, catCount, dataArea,
17          domainEdge);
18      lineY2 = rangeAxis.valueToJava2D(this.value2, dataArea, rangeEdge);
19  }
```

M = {1,5,14}

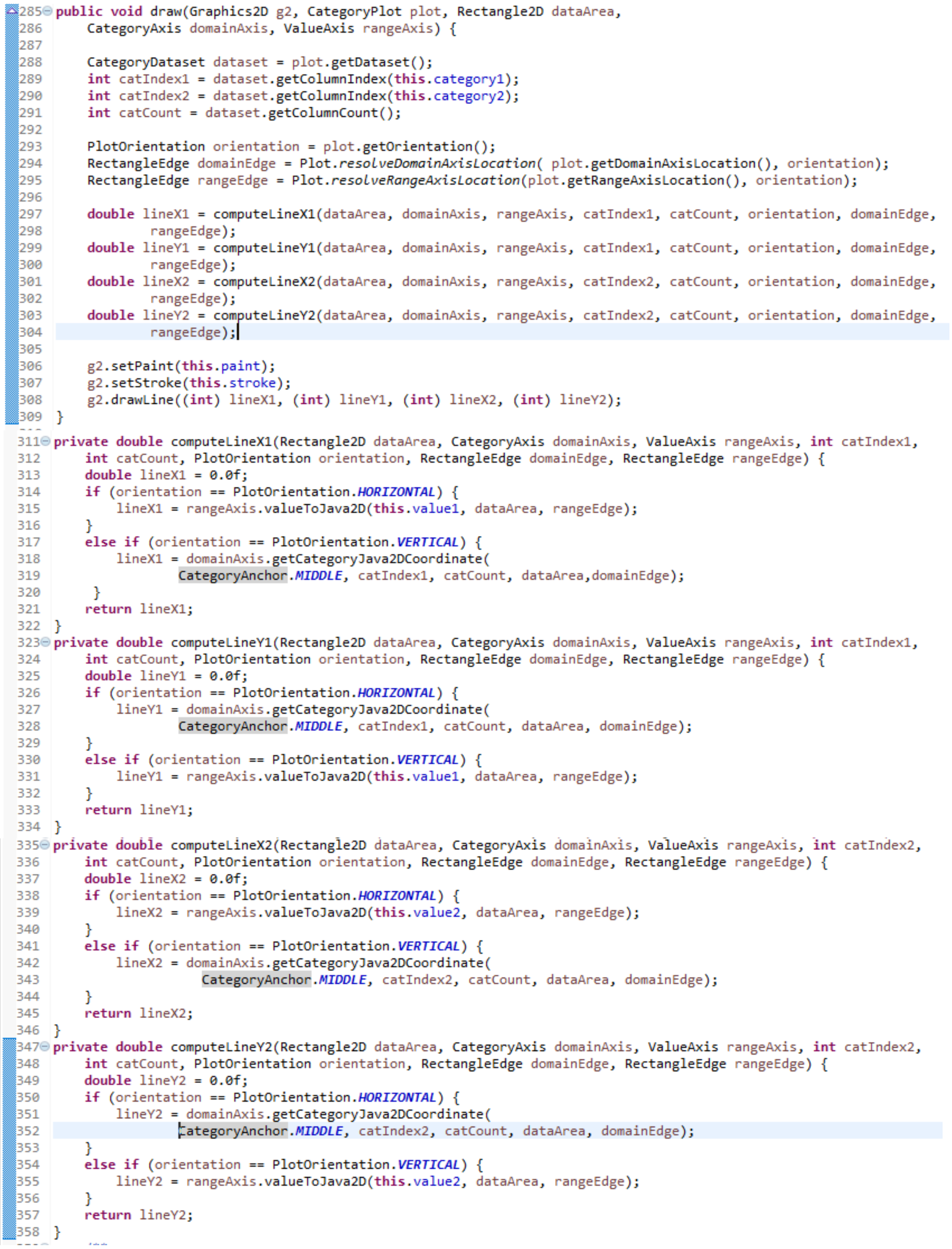
the result after step 2:



```
public void draw(Graphics2D g2, CategoryPlot plot, Rectangle2D dataArea,
                 CategoryAxis domainAxis, ValueAxis rangeAxis) {

    CategoryDataset dataset = plot.getDataset();
    int catIndex1 = dataset.getColumnIndex(this.category1);
    int catIndex2 = dataset.getColumnIndex(this.category2);
    int catCount = dataset.getColumnCount();

    PlotOrientation orientation = plot.getOrientation();
    RectangleEdge domainEdge = Plot.resolveDomainAxisLocation(
                    plot.getDomainAxisLocation(), orientation);
    RectangleEdge rangeEdge = Plot.resolveRangeAxisLocation(
                    plot.getRangeAxisLocation(), orientation);

    double lineX1 = computeLineX1(dataArea, domainAxis, rangeAxis, catIndex1, catCount, orientation, domainEdge,
                rangeEdge);
    double lineY1 = computeLineY1(dataArea, domainAxis, rangeAxis, catIndex1, catCount, orientation, domainEdge,
            rangeEdge);

    double lineX2 = 0.0f;
    double lineY2 = 0.0f;
    if (orientation == PlotOrientation.HORIZONTAL) {
        lineY2 = domainAxis.getCategoryJava2DCoordinate(
            CategoryAnchor.MIDDLE, catIndex2, catCount, dataArea,
            domainEdge);
        lineX2 = rangeAxis.valueToJava2D(this.value2, dataArea, rangeEdge);
    }
    else if (orientation == PlotOrientation.VERTICAL) {
        lineX2 = domainAxis.getCategoryJava2DCoordinate(
            CategoryAnchor.MIDDLE, catIndex2, catCount, dataArea,
            domainEdge);
        lineY2 = rangeAxis.valueToJava2D(this.value2, dataArea, rangeEdge);
    }

    g2.setPaint(this.paint);
    g2.setStroke(this.stroke);
    g2.drawLine((int) lineX1, (int) lineY1, (int) lineX2, (int) lineY2);
}

private double computeLineX1(Rectangle2D dataArea, CategoryAxis domainAxis, ValueAxis rangeAxis, int catIndex1,
        int catCount, PlotOrientation orientation, RectangleEdge domainEdge, RectangleEdge rangeEdge) {
        double lineX1 = 0.0f;
        if (orientation == PlotOrientation.HORIZONTAL) {
        lineX1 = rangeAxis.valueToJava2D(this.value1, dataArea, rangeEdge);
        }
        else if (orientation == PlotOrientation.VERTICAL) {
        lineX1 = domainAxis.getCategoryJava2DCoordinate(
                CategoryAnchor.MIDDLE, catIndex1, catCount, dataArea,
                domainEdge);
            }
        return lineX1;
    }

private double computeLineY1(Rectangle2D dataArea, CategoryAxis domainAxis, ValueAxis rangeAxis, int catIndex1,
        int catCount, PlotOrientation orientation, RectangleEdge domainEdge, RectangleEdge rangeEdge) {
        double lineY1 = 0.0f;
        if (orientation == PlotOrientation.HORIZONTAL) {
            lineY1 = domainAxis.getCategoryJava2DCoordinate(
                CategoryAnchor.MIDDLE, catIndex1, catCount, dataArea,
                domainEdge);
        }
        else if (orientation == PlotOrientation.VERTICAL) {
            lineY1 = rangeAxis.valueToJava2D(this.value1, dataArea, rangeEdge);
        }
        return lineY1;
    }
```

In the same way we performed step3 and step4 for the other variables.

The final result will be :



```java
public void draw(Graphics2D g2, CategoryPlot plot, Rectangle2D dataArea,
    CategoryAxis domainAxis, ValueAxis rangeAxis) {

    CategoryDataset dataset = plot.getDataset();
    int catIndex1 = dataset.getColumnIndex(this.category1);
    int catIndex2 = dataset.getColumnIndex(this.category2);
    int catCount = dataset.getColumnCount();

    PlotOrientation orientation = plot.getOrientation();
    RectangleEdge domainEdge = Plot.resolveDomainAxisLocation( plot.getDomainAxisLocation(), orientation);
    RectangleEdge rangeEdge = Plot.resolveRangeAxisLocation(plot.getRangeAxisLocation(), orientation);

    double lineX1 = computeLineX1(dataArea, domainAxis, rangeAxis, catIndex1, catCount, orientation, domainEdge,
            rangeEdge);
    double lineY1 = computeLineY1(dataArea, domainAxis, rangeAxis, catIndex1, catCount, orientation, domainEdge,
            rangeEdge);
    double lineX2 = computeLineX2(dataArea, domainAxis, rangeAxis, catIndex2, catCount, orientation, domainEdge,
            rangeEdge);
    double lineY2 = computeLineY2(dataArea, domainAxis, rangeAxis, catIndex2, catCount, orientation, domainEdge,
            rangeEdge);

    g2.setPaint(this.paint);
    g2.setStroke(this.stroke);
    g2.drawLine((int) lineX1, (int) lineY1, (int) lineX2, (int) lineY2);
}
private double computeLineX1(Rectangle2D dataArea, CategoryAxis domainAxis, ValueAxis rangeAxis, int catIndex1,
    int catCount, PlotOrientation orientation, RectangleEdge domainEdge, RectangleEdge rangeEdge) {
    double lineX1 = 0.0f;
    if (orientation == PlotOrientation.HORIZONTAL) {
        lineX1 = rangeAxis.valueToJava2D(this.value1, dataArea, rangeEdge);
    }
    else if (orientation == PlotOrientation.VERTICAL) {
        lineX1 = domainAxis.getCategoryJava2DCoordinate(
                CategoryAnchor.MIDDLE, catIndex1, catCount, dataArea,domainEdge);
     }
    return lineX1;
}
private double computeLineY1(Rectangle2D dataArea, CategoryAxis domainAxis, ValueAxis rangeAxis, int catIndex1,
    int catCount, PlotOrientation orientation, RectangleEdge domainEdge, RectangleEdge rangeEdge) {
    double lineY1 = 0.0f;
    if (orientation == PlotOrientation.HORIZONTAL) {
        lineY1 = domainAxis.getCategoryJava2DCoordinate(
                CategoryAnchor.MIDDLE, catIndex1, catCount, dataArea, domainEdge);
    }
    else if (orientation == PlotOrientation.VERTICAL) {
        lineY1 = rangeAxis.valueToJava2D(this.value1, dataArea, rangeEdge);
    }
    return lineY1;
}
private double computeLineX2(Rectangle2D dataArea, CategoryAxis domainAxis, ValueAxis rangeAxis, int catIndex2,
    int catCount, PlotOrientation orientation, RectangleEdge domainEdge, RectangleEdge rangeEdge) {
    double lineX2 = 0.0f;
    if (orientation == PlotOrientation.HORIZONTAL) {
        lineX2 = rangeAxis.valueToJava2D(this.value2, dataArea, rangeEdge);
    }
    else if (orientation == PlotOrientation.VERTICAL) {
        lineX2 = domainAxis.getCategoryJava2DCoordinate(
                   CategoryAnchor.MIDDLE, catIndex2, catCount, dataArea, domainEdge);
    }
    return lineX2;
}
private double computeLineY2(Rectangle2D dataArea, CategoryAxis domainAxis, ValueAxis rangeAxis, int catIndex2,
    int catCount, PlotOrientation orientation, RectangleEdge domainEdge, RectangleEdge rangeEdge) {
    double lineY2 = 0.0f;
    if (orientation == PlotOrientation.HORIZONTAL) {
        lineY2 = domainAxis.getCategoryJava2DCoordinate(
                CategoryAnchor.MIDDLE, catIndex2, catCount, dataArea, domainEdge);
    }
    else if (orientation == PlotOrientation.VERTICAL) {
        lineY2 = rangeAxis.valueToJava2D(this.value2, dataArea, rangeEdge);
    }
    return lineY2;
}
```

Conclusion:

In conclusion, the draw method is simpler then before and easier to maintain.

Now we have four helper methods and we can reuse them in other places in the project, although, the runtime might be larger .

Looking back, in this specific code one can argue that it would be better to leave the code as it is because those four variables have the same purpose (coordinates) and should be compute together, nonetheless, we decided to do so for the purpose of learning as well.

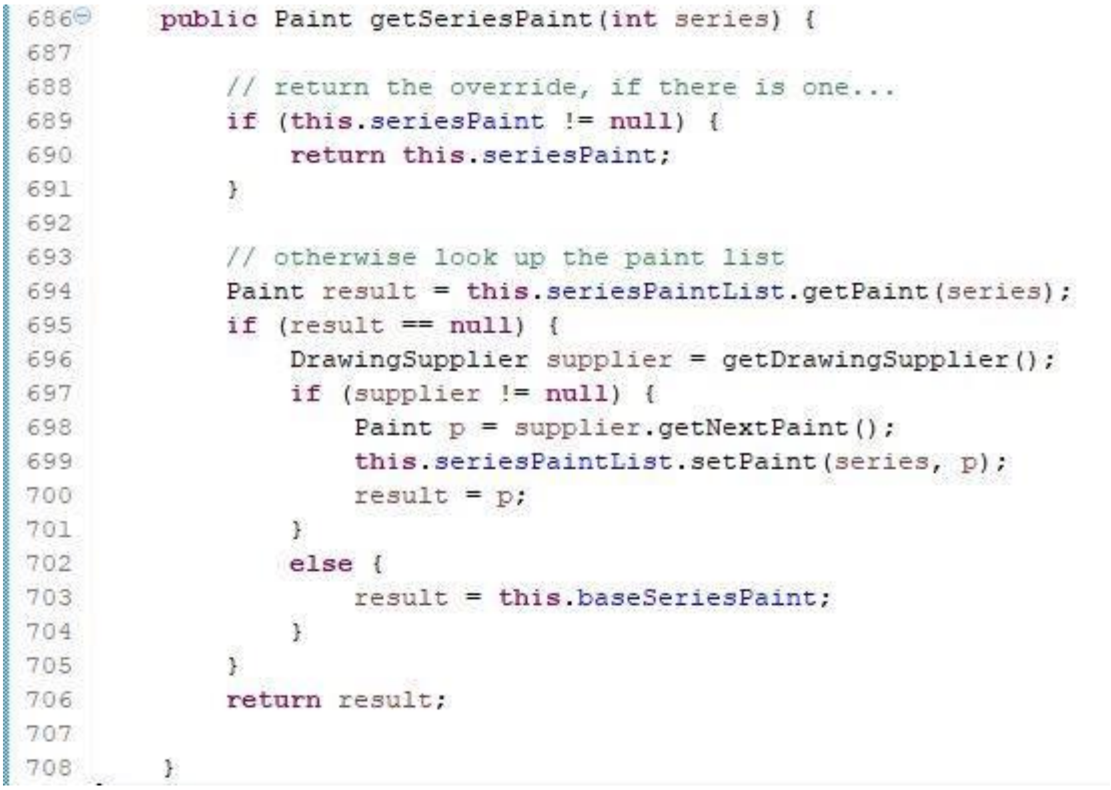
SQFM:

We chose a method called "getSeriesPaint" from an open source project which can be found at:

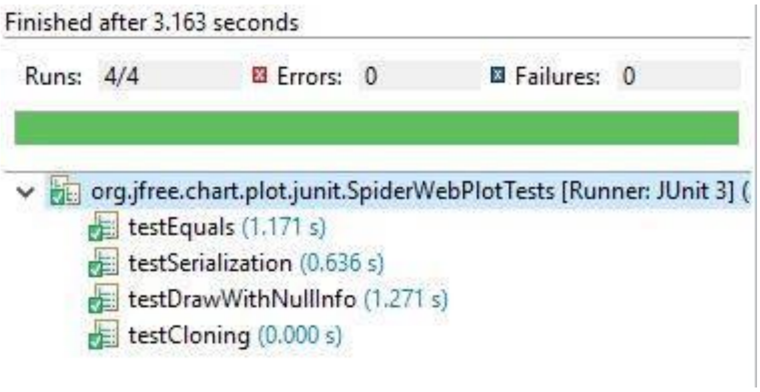
https://github.com/alwaqfi/jfreechart-1.0.10/blob/master/src/org/jfree/chart/plot/SpiderWebPlot.java

jfreechart-1.0.10/src/org/jfree/chart/plot/**SpiderWebPlot.java**

Original code:



```
686     public Paint getSeriesPaint(int series) {
687 
688         // return the override, if there is one...
689         if (this.seriesPaint != null) {
690             return this.seriesPaint;
691         }
692 
693         // otherwise look up the paint list
694         Paint result = this.seriesPaintList.getPaint(series);
695         if (result == null) {
696             DrawingSupplier supplier = getDrawingSupplier();
697             if (supplier != null) {
698                 Paint p = supplier.getNextPaint();
699                 this.seriesPaintList.setPaint(series, p);
700                 result = p;
701             }
702             else {
703                 result = this.baseSeriesPaint;
704             }
705         }
706         return result;
707 
708     }
```



```
Finished after 3.163 seconds

Runs: 4/4      ☒ Errors: 0      ☒ Failures: 0

▾ org.jfree.chart.plot.junit.SpiderWebPlotTests [Runner: JUnit 3] (
      testEquals (1.171 s)
      testSerialization (0.636 s)
      testDrawWithNullInfo (1.271 s)
      testCloning (0.000 s)
```

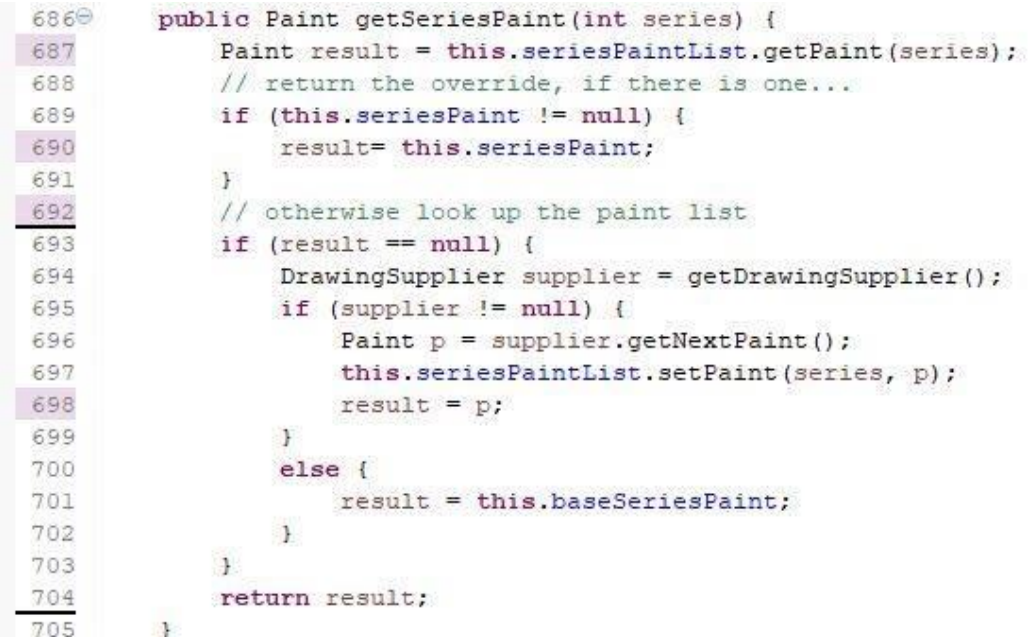
As we can see this is a non- void method that have side effect on object field (line 669: this.seriesPaintList.setPaint(series,p);)

Therefore, we apply sperate query from modifier refactoring.

Step 1:

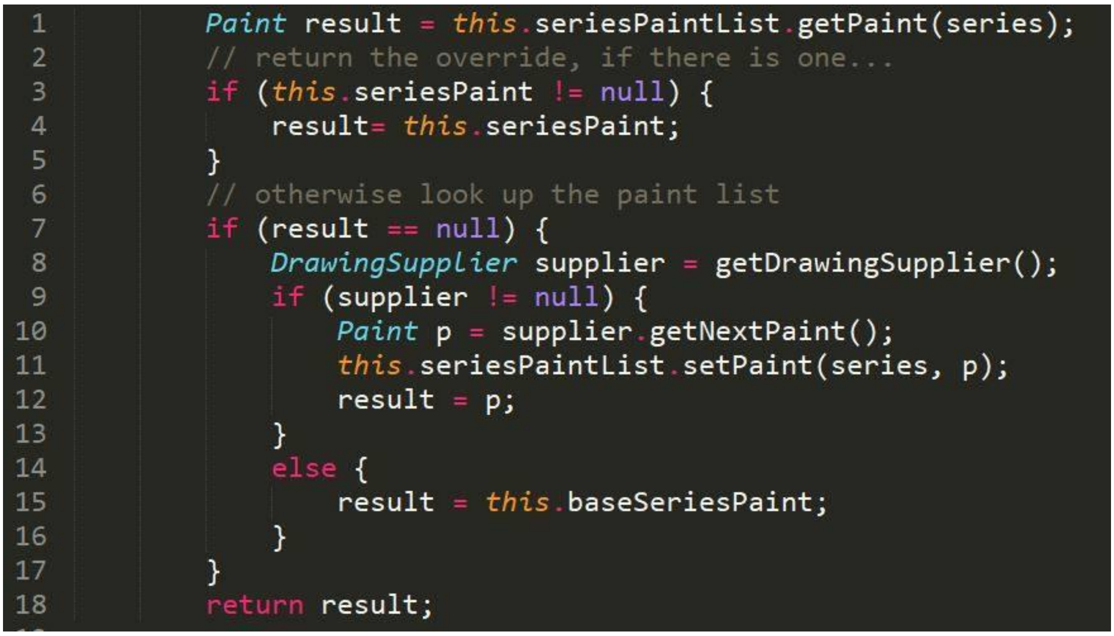
In this step we prepared the code for Sliding algorithm by adjusting the code to have a single return value

* this attempt is a wrong one that might cause problems.



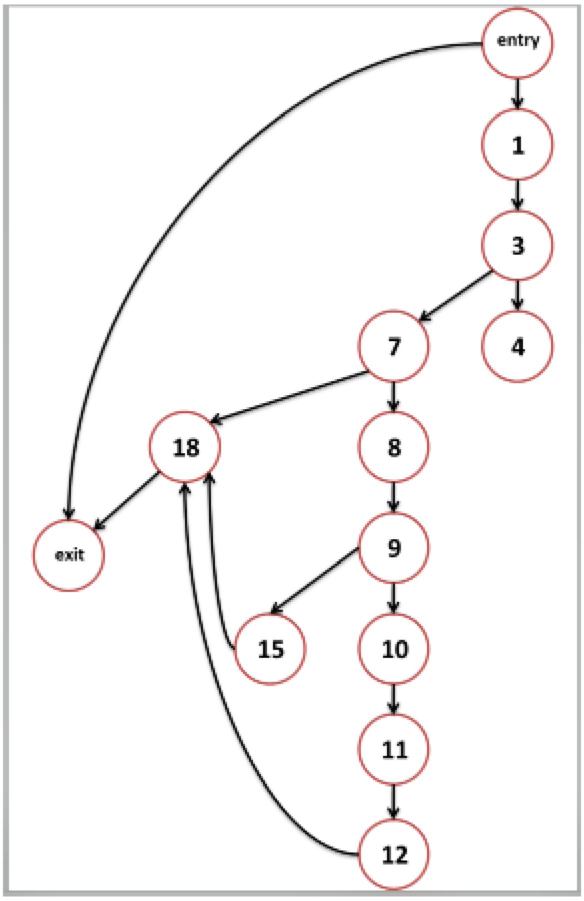
```
686     public Paint getSeriesPaint(int series) {
687         Paint result = this.seriesPaintList.getPaint(series);
688         // return the override, if there is one...
689         if (this.seriesPaint != null) {
690             result= this.seriesPaint;
691         }
692         // otherwise look up the paint list
693         if (result == null) {
694             DrawingSupplier supplier = getDrawingSupplier();
695             if (supplier != null) {
696                 Paint p = supplier.getNextPaint();
697                 this.seriesPaintList.setPaint(series, p);
698                 result = p;
699             }
700             else {
701                 result = this.baseSeriesPaint;
702             }
703         }
704         return result;
705     }
```

We chose the block of code from lines 687-704, for convenience we numbered those lines from 1-18.



```
1           Paint result = this.seriesPaintList.getPaint(series);
2           // return the override, if there is one...
3           if (this.seriesPaint != null) {
4               result= this.seriesPaint;
5           }
6           // otherwise look up the paint list
7           if (result == null) {
8               DrawingSupplier supplier = getDrawingSupplier();
9               if (supplier != null) {
10                  Paint p = supplier.getNextPaint();
11                  this.seriesPaintList.setPaint(series, p);
12                  result = p;
13              }
14              else {
15                  result = this.baseSeriesPaint;
16              }
17          }
18          return result;
```

Build the CFG:



Build the PDG:

Before we can perform the bucketing, we need to compute the PDG.

*in case of a non- void method we address the "return line" as a definition to a variable called ret_val.

**we assume that the method "getNextPaint" change the supplier object thus we add flow dependence (10, exit).

*** we assume that the method "getPaint" is only a getter and doesn't change anything.

Edge	type
(Entry,1)	**Control**
(Entry,3)	**Control**
(Entry,7)	**Control**
(Entry,18)	**Control**
(3,4)	**Control**
(7,8)	**Control**
(7,9)	**Control**
(9,10)	**Control**
(9,11)	**Control**
(9,12)	**Control**
(9,15)	**Control**

Edge	type	Vars
(1, Exit)	flow	**{result}**
(4,7)	flow	**{result}**
(1,7)	flow	**{result}**
(1,18)	flow	**{result}**
(4,18)	flow	**{result}**
(12,18)	flow	**{result}**
(15,18)	flow	**{result}**
(4, Exit)	flow	**{result}**
(12, Exit)	flow	**{result}**
(15, Exit)	flow	**{result}**
(entry,3)	flow	**{this.seriesPaint}**
(entry,4)	flow	**{this.seriesPaint}**
(8,9)	flow	**{ supplier }**
(8,10)	flow	**{supplier , supplier.*}**
(8,exit)	flow	**{supplier}**
(10,11)	flow	**{p}**
(10,12)	flow	**{p}**
(10,exit)	flow	**{p, supplier, supplier.*}**
(entry,11)	flow	**{ series }**
(entry,1)	flow	**{ this.seriesPaintList.* ,this.seriesPaintList ,series}**
(11,exit)	flow	**{ this.seriesPaintList, this.seriesPaintList .* }**
(entry,15)	flow	**{ this.baseSeriesPaint }**
(18,exit)	flow	**{ ret_val }**

Edge	type	Vars
(7,15)	anti	**{ result }**
(7,12)	anti	**{ result}**
(11,11)	anti	**{ this.seriesPaintList , this.seriesPaintList .* }**
(1,11)	anti	**{ this.seriesPaintList , this.seriesPaintList .* }**
(9,10)	anti	**{ supplier }**

Step 2:

We apply sliding algorithm on V={result}, the Slice will be the Query and the Co-Slice will be the Modifier.

Slice:

To do the slice we remove flow dependencies to exit of any variable other than "result".

Now we do back tracking starting from exit node up, using control and flow dependencies.

Slice(exit) = {entry, 1, 3, 4, 7, 8, 9, 10, 12, 15, exit}

Co- Slice:

To do the Co-Slice we remove flow dependence created by the variable "result" (y, x) such that there is no anti dependence (x ,z) created by the variable "result".

Those are the edges that left:

Edge	type
(Entry,1)	**Control**
(Entry,3)	**Control**
(Entry,7)	**Control**
(Entry,18)	**Control**
(3,4)	**Control**
(7,8)	**Control**
(7,9)	**Control**
(9,10)	**Control**
(9,11)	**Control**
(9,12)	**Control**
(9,15)	**Control**

Edge	type	Vars
(4,7)	flow	**{result}**
(1,7)	flow	**{result}**
(entry,3)	flow	**{this.seriesPaint}**
(entry,4)	flow	**{this.seriesPaint}**
(8,9)	flow	**{ supplier }**
(8,10)	flow	**{supplier , supplier.*}**
(8,exit)	flow	**{supplier}**
(10,11)	flow	**{p}**
(10,12)	flow	**{p}**
(10,exit)	flow	**{p, supplier, supplier.*}**
(entry,11)	flow	**{ this.seriesPaintList.* ,this.seriesPaintList ,series}**
(entry,1)	flow	**{ this.seriesPaintList.* ,this.seriesPaintList ,series}**
(11,exit)	flow	**{ this.seriesPaintList, this.seriesPaintList .* }**
(entry,15)	flow	**{ this.baseSeriesPaint }**
(18,exit)	flow	**{ ret_val }**

Now we do back tracking starting from exit node up, using control and flow dependencies.

Co-Slice (exit) = {entry ,1, 3, 4, 7, 8, 9, 10, 11, 18}

Compensations checking:

Pen1: The value of the extracted variable could change in the Co-Slice.

We noticed that node 1, 4 , ,existing in Co-Slice, define variable "result" therefore, **pen1={[1,result], [4, result]}**.

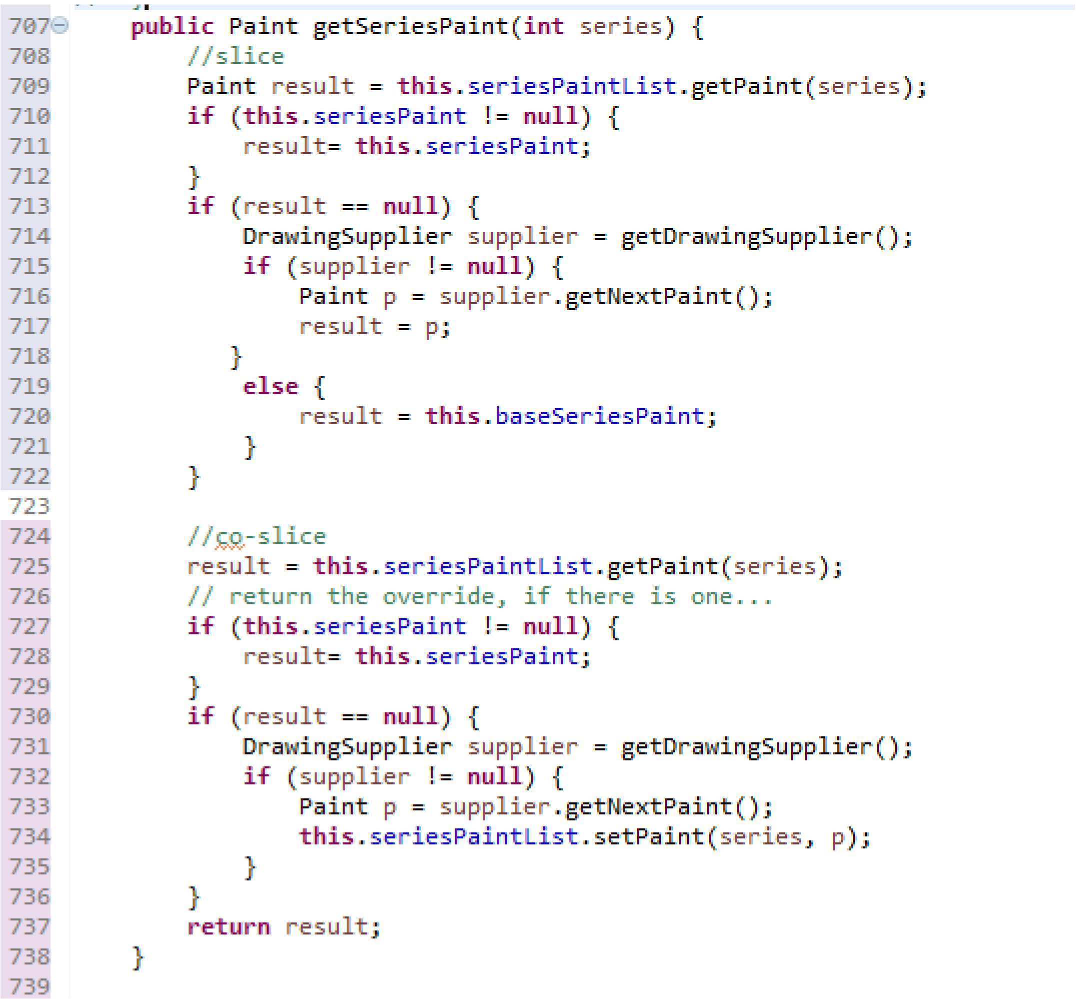
Pen2: there exists a usage in the Co-Slice of a non-final value of the extracted variable.

We noticed that node 7, existing in Co-Slice, uses variable "result" and there is anti-dependence from node 7 therefore, **pen2={[7,result]}**.

Pen3: the slice could change the value of unextracted variable, that its initial value is required at the co-slice.

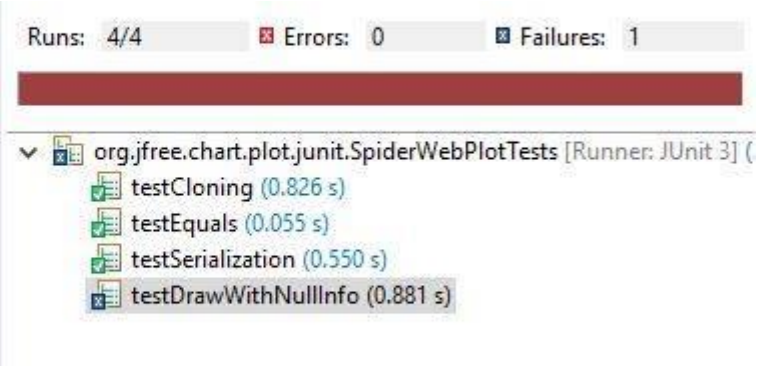
There isn't Slice's nodes defining variables causing flow dependencies from entry to Co-Slice's nodes, therefore **pen3 = {}**.

The result for our first attempt will be:



```
    public Paint getSeriesPaint(int series) {
        //slice
        Paint result = this.seriesPaintList.getPaint(series);
        if (this.seriesPaint != null) {
            result= this.seriesPaint;
        }
        if (result == null) {
            DrawingSupplier supplier = getDrawingSupplier();
            if (supplier != null) {
                Paint p = supplier.getNextPaint();
                result = p;
           }
            else {
                result = this.baseSeriesPaint;
            }
        }

        //co-slice
        result = this.seriesPaintList.getPaint(series);
        // return the override, if there is one...
        if (this.seriesPaint != null) {
            result= this.seriesPaint;
        }
        if (result == null) {
            DrawingSupplier supplier = getDrawingSupplier();
            if (supplier != null) {
                Paint p = supplier.getNextPaint();
                this.seriesPaintList.setPaint(series, p);
            }
        }
        return result;
    }
```

As we can see, this result is wrong and not even efficient.

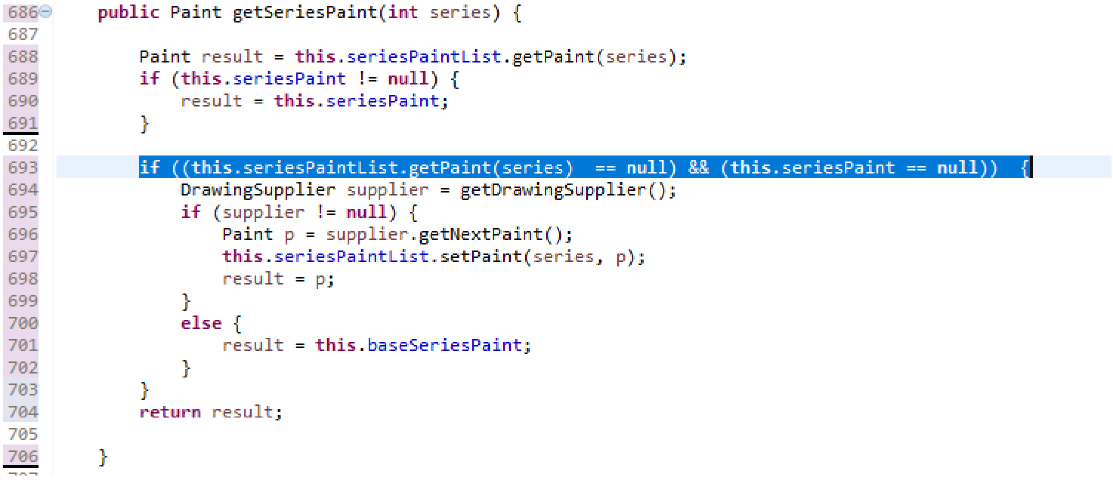


```
Runs: 4/4    Errors: 0    Failures: 1

org.jfree.chart.plot.junit.SpiderWebPlotTests [Runner: JUnit 3] (
    testCloning (0.826 s)
    testEquals (0.055 s)
    testSerialization (0.550 s)
    testDrawWithNullInfo (0.881 s)
```

Second attempt – Sliding:

Step1:

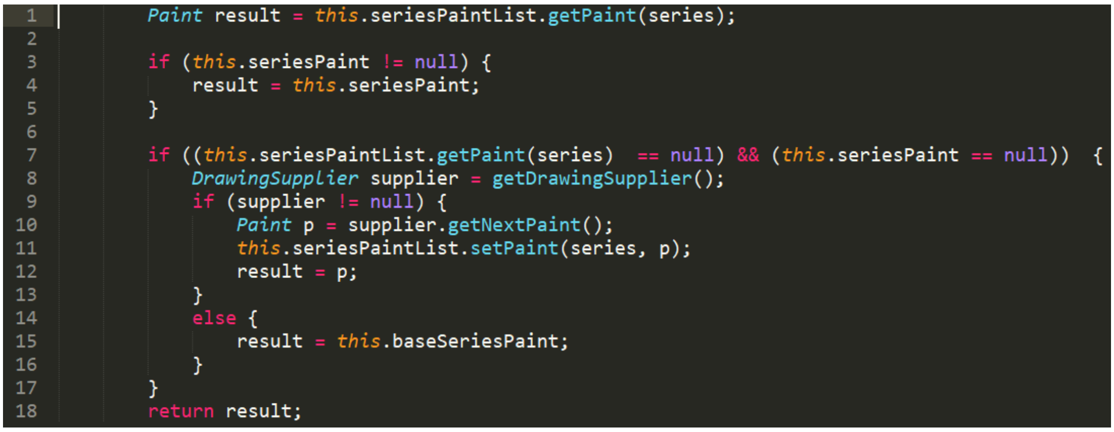
To correct the problem from the first attempt we will perform this change:



```
686     public Paint getSeriesPaint(int series) {
687 
688         Paint result = this.seriesPaintList.getPaint(series);
689         if (this.seriesPaint != null) {
690             result = this.seriesPaint;
691         }
692 
693         if ((this.seriesPaintList.getPaint(series)  == null) && (this.seriesPaint == null))  {
694             DrawingSupplier supplier = getDrawingSupplier();
695             if (supplier != null) {
696                 Paint p = supplier.getNextPaint();
697                 this.seriesPaintList.setPaint(series, p);
698                 result = p;
699             }
700             else {
701                 result = this.baseSeriesPaint;
702             }
703         }
704         return result;
705 
706     }
```

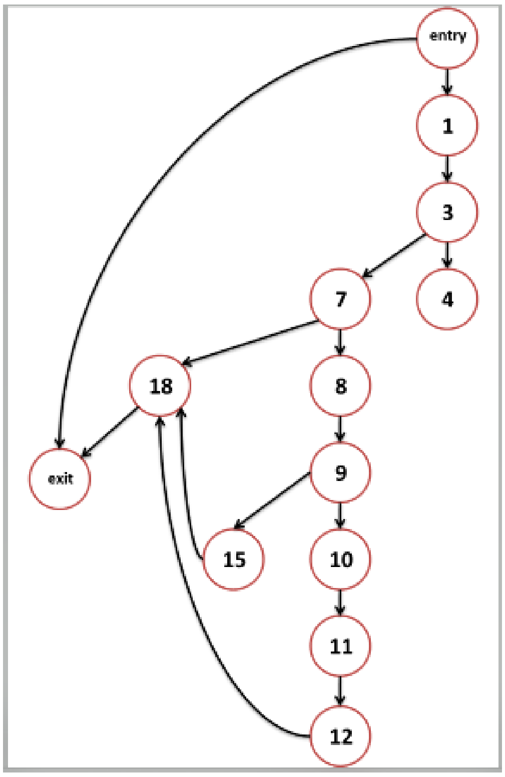
This change will keep the correctness of the code and now we do the Sliding algorithm again.

We chose the block of code from lines 688-704, for convenience we numbered those lines from 1-18 .



```
1           Paint result = this.seriesPaintList.getPaint(series);
2   
3           if (this.seriesPaint != null) {
4               result = this.seriesPaint;
5           }
6   
7           if ((this.seriesPaintList.getPaint(series)  == null) && (this.seriesPaint == null))  {
8               DrawingSupplier supplier = getDrawingSupplier();
9               if (supplier != null) {
10                  Paint p = supplier.getNextPaint();
11                  this.seriesPaintList.setPaint(series, p);
12                  result = p;
13              }
14              else {
15                  result = this.baseSeriesPaint;
16              }
17          }
18          return result;
```

Build the CFG:



Build the PDG:

Edge	type
(Entry,1)	**Control**
(Entry,3)	**Control**
(Entry,7)	**Control**
(Entry,18)	**Control**
(3,4)	**Control**
(7,8)	**Control**
(7,9)	**Control**
(9,10)	**Control**
(9,11)	**Control**
(9,12)	**Control**
(9,15)	**Control**

Edge	type	Vars
(11,11)	anti	**{ this.seriesPaintList , this.seriesPaintList .* }**
(1,11)	anti	**{ this.seriesPaintList , this.seriesPaintList .* }**
(9,10)	anti	**{ supplier }**

Edge	type	Vars
(1,Exit)	flow	**{result}**
(1,18)	flow	**{result}**
(4,18)	flow	**{result}**
(12,18)	flow	**{result}**
(15,18)	flow	**{result}**
(4,Exit)	flow	**{result}**
(12,Exit)	flow	**{result}**
(15,Exit)	flow	**{result}**
(entry,3)	flow	**{this.seriesPaint}**
(entry,4)	flow	**{this.seriesPaint}**
(entry,7)	flow	**{this.seriesPaint}**
(8,9)	flow	**{ supplier }**
(8,10)	flow	**{supplier , supplier.*}**
(8,exit)	flow	**{supplier}**
(10,11)	flow	**{p}**
(10,12)	flow	**{p}**
(10,exit)	flow	**{p, supplier, supplier.*}**
(entry,11)	flow	**{ this.seriesPaintList.* ,this.seriesPaintList ,series}**
(entry,1)	flow	**{ this.seriesPaintList.* ,this.seriesPaintList ,series}**
(entry,7)	flow	**{ this.seriesPaintList.* ,this.seriesPaintList ,series}**
(11,exit)	flow	**{ this.seriesPaintList, this.seriesPaintList .* }**
(entry,15)	flow	**{ this.baseSeriesPaint }**
(18,exit)	flow	**{ ret_val }**

Slice = {entry, 1, 3, 4, 7, 8, 9, 10, 12, 15, exit}

Co-Slice = {entry, 7, 8, 9, 10, 11, 18, exit}

Compensations checking:

Pen1: The value of the extracted variable could change in the Co-Slice.

There is no definition to variable "result" in any Co-Slice's nodes, therefore **pen1 = {}**.

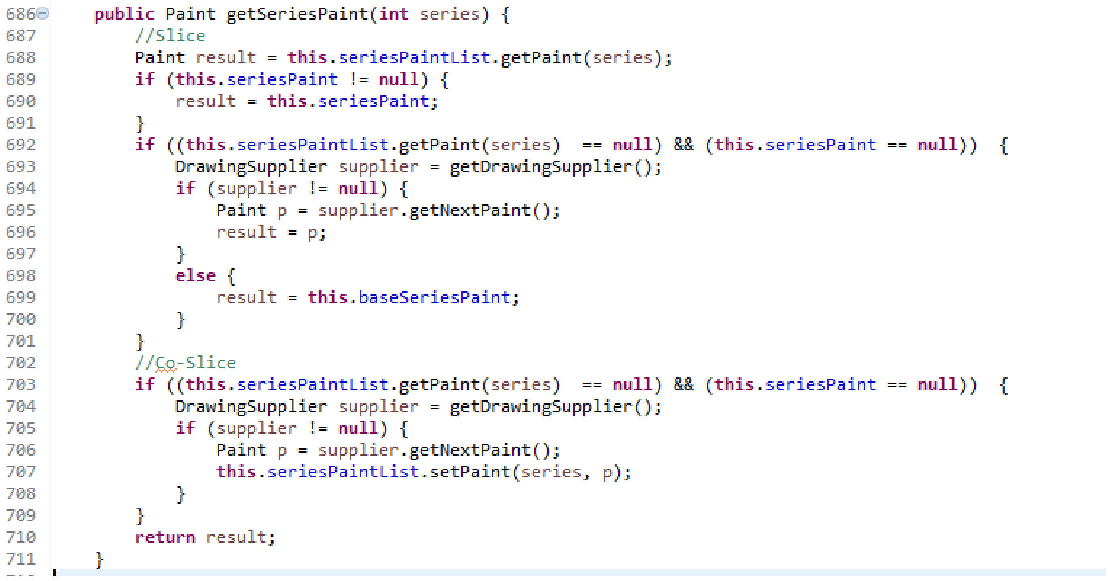
Pen2: there exists a usage in the Co-Slice of a non-final value of the extracted variable.

There is no usage to variable "result" in any Co-Slice's nodes, therefore **pen2 = {}**.

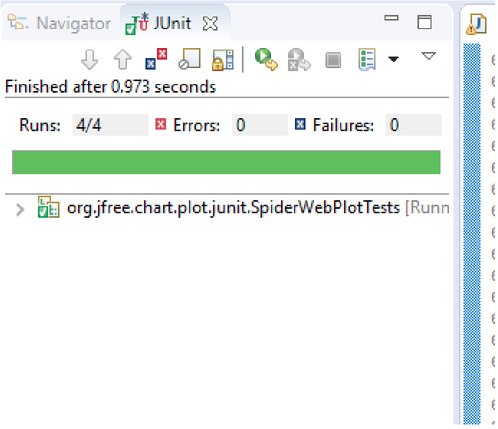
Pen3: the slice could change the value of unextracted variable, that its initial value is required at the co-slice.

There isn't Slice's nodes defining variables causing flow dependencies from entry to Co-Slice's nodes, therefore **pen3 = {}**.

The result will be:

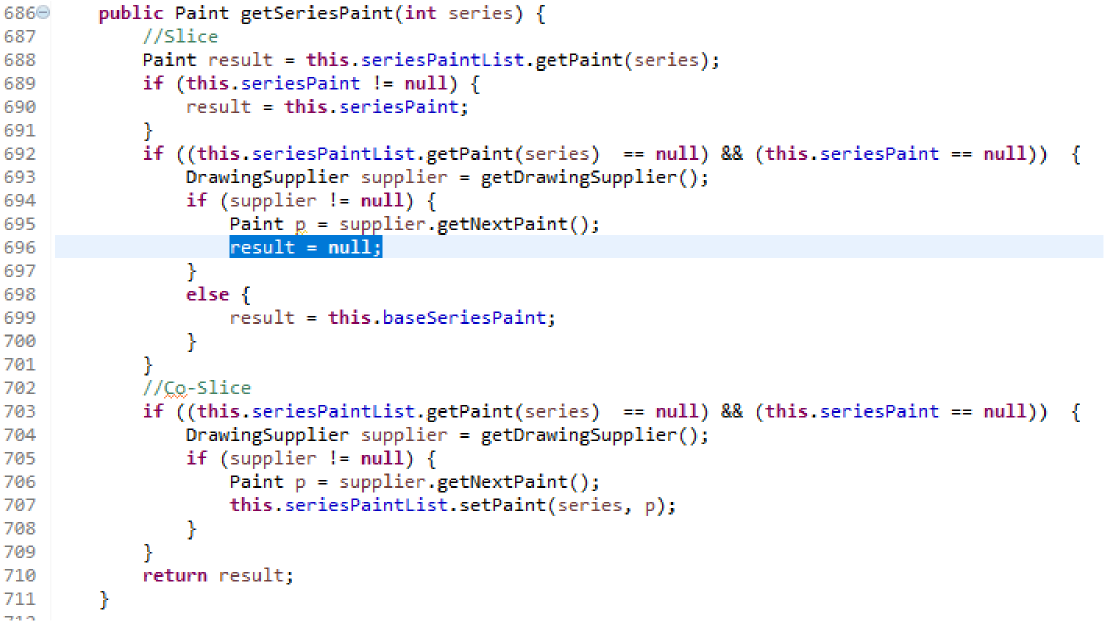


```
    public Paint getSeriesPaint(int series) {
        //Slice
        Paint result = this.seriesPaintList.getPaint(series);
        if (this.seriesPaint != null) {
            result = this.seriesPaint;
        }
        if ((this.seriesPaintList.getPaint(series)  == null) && (this.seriesPaint == null))  {
            DrawingSupplier supplier = getDrawingSupplier();
            if (supplier != null) {
                Paint p = supplier.getNextPaint();
                result = p;
            }
            else {
                result = this.baseSeriesPaint;
            }
        }
        //Co-Slice
        if ((this.seriesPaintList.getPaint(series)  == null) && (this.seriesPaint == null))  {
            DrawingSupplier supplier = getDrawingSupplier();
            if (supplier != null) {
                Paint p = supplier.getNextPaint();
                this.seriesPaintList.setPaint(series, p);
            }
        }
        return result;
    }
```

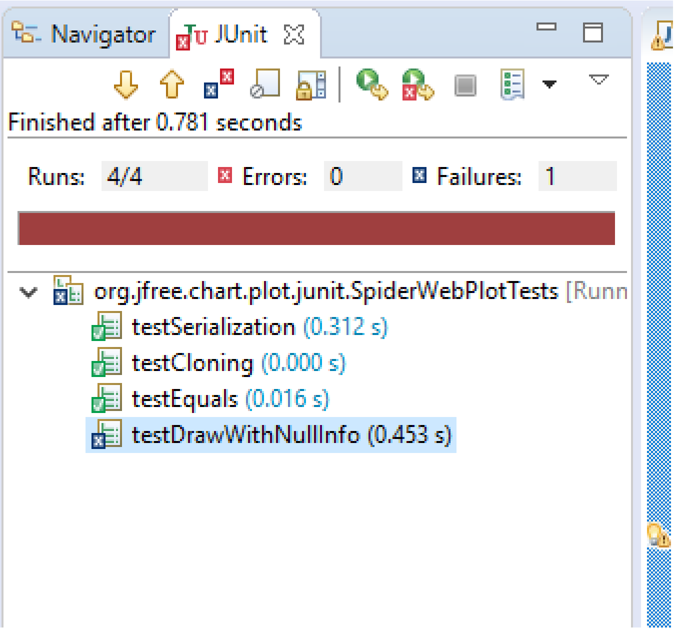


```
Finished after 0.973 seconds

Runs:  4/4      Errors:  0      Failures:  0

org.jfree.chart.plot.junit.SpiderWebPlotTests [Runn
```

Let's inject a piece of code that will cause a failure due to intentional mistake.



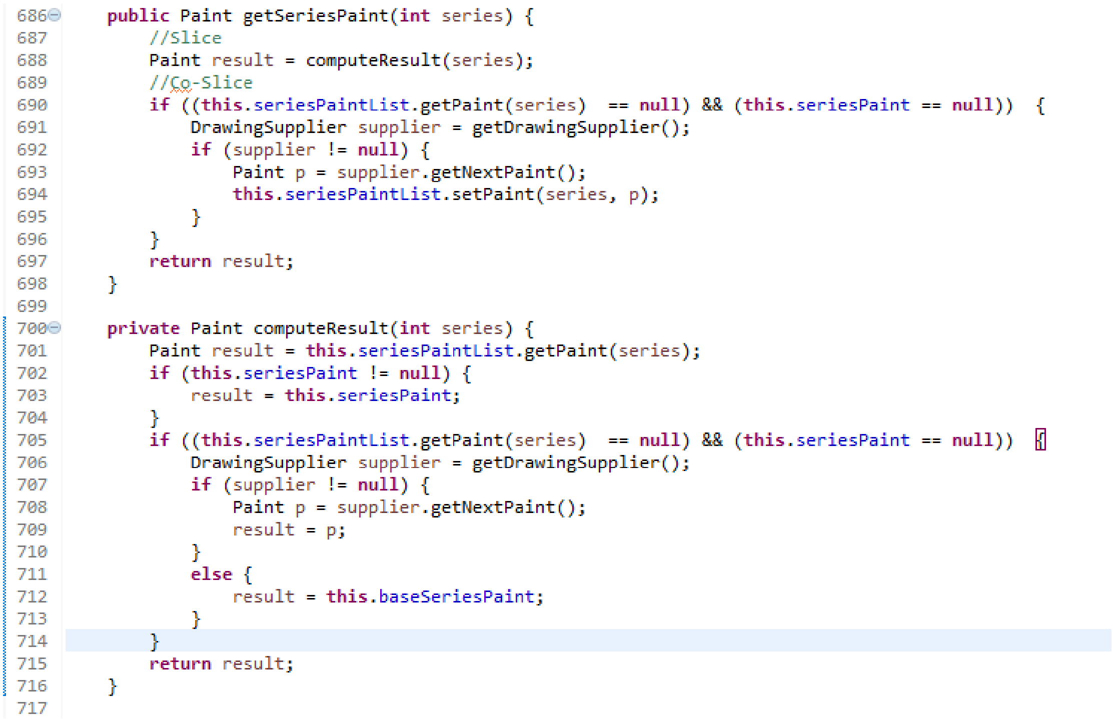
```
    public Paint getSeriesPaint(int series) {
        //Slice
        Paint result = this.seriesPaintList.getPaint(series);
        if (this.seriesPaint != null) {
            result = this.seriesPaint;
        }
        if ((this.seriesPaintList.getPaint(series)  == null) && (this.seriesPaint == null))  {
            DrawingSupplier supplier = getDrawingSupplier();
            if (supplier != null) {
                Paint p = supplier.getNextPaint();
                result = null;
            }
            else {
                result = this.baseSeriesPaint;
            }
        }
        //Co-Slice
        if ((this.seriesPaintList.getPaint(series)  == null) && (this.seriesPaint == null))  {
            DrawingSupplier supplier = getDrawingSupplier();
            if (supplier != null) {
                Paint p = supplier.getNextPaint();
                this.seriesPaintList.setPaint(series, p);
            }
        }
        return result;
    }
```



```
Finished after 0.781 seconds

Runs:  4/4    Errors:  0    Failures:  1

org.jfree.chart.plot.junit.SpiderWebPlotTests [Runn
    testSerialization (0.312 s)
    testCloning (0.000 s)
    testEquals (0.016 s)
    testDrawWithNullInfo (0.453 s)
```

Step3:

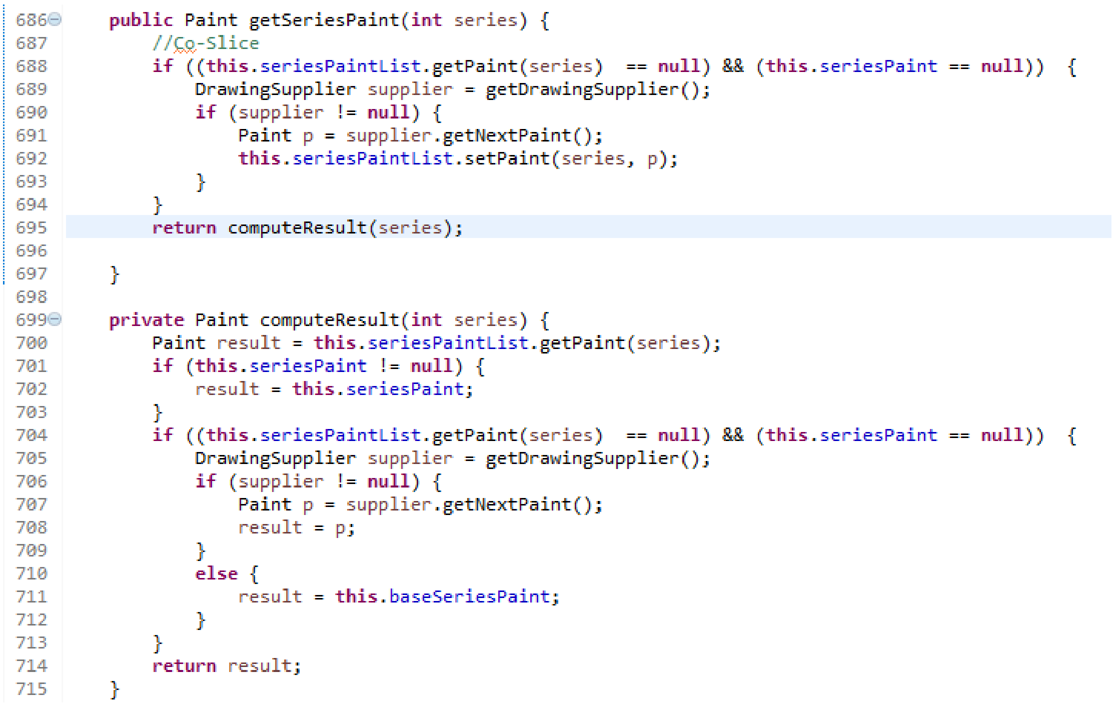
Extract method to Query.



```
public Paint getSeriesPaint(int series) {
    //Slice
    Paint result = computeResult(series);
    //Co-Slice
    if ((this.seriesPaintList.getPaint(series)  == null) && (this.seriesPaint == null))  {
        DrawingSupplier supplier = getDrawingSupplier();
        if (supplier != null) {
            Paint p = supplier.getNextPaint();
            this.seriesPaintList.setPaint(series, p);
        }
    }
    return result;
}

private Paint computeResult(int series) {
    Paint result = this.seriesPaintList.getPaint(series);
    if (this.seriesPaint != null) {
        result = this.seriesPaint;
    }
    if ((this.seriesPaintList.getPaint(series)  == null) && (this.seriesPaint == null))  {
        DrawingSupplier supplier = getDrawingSupplier();
        if (supplier != null) {
            Paint p = supplier.getNextPaint();
            result = p;
        }
        else {
            result = this.baseSeriesPaint;
        }
    }
    return result;
}
```

Step4:

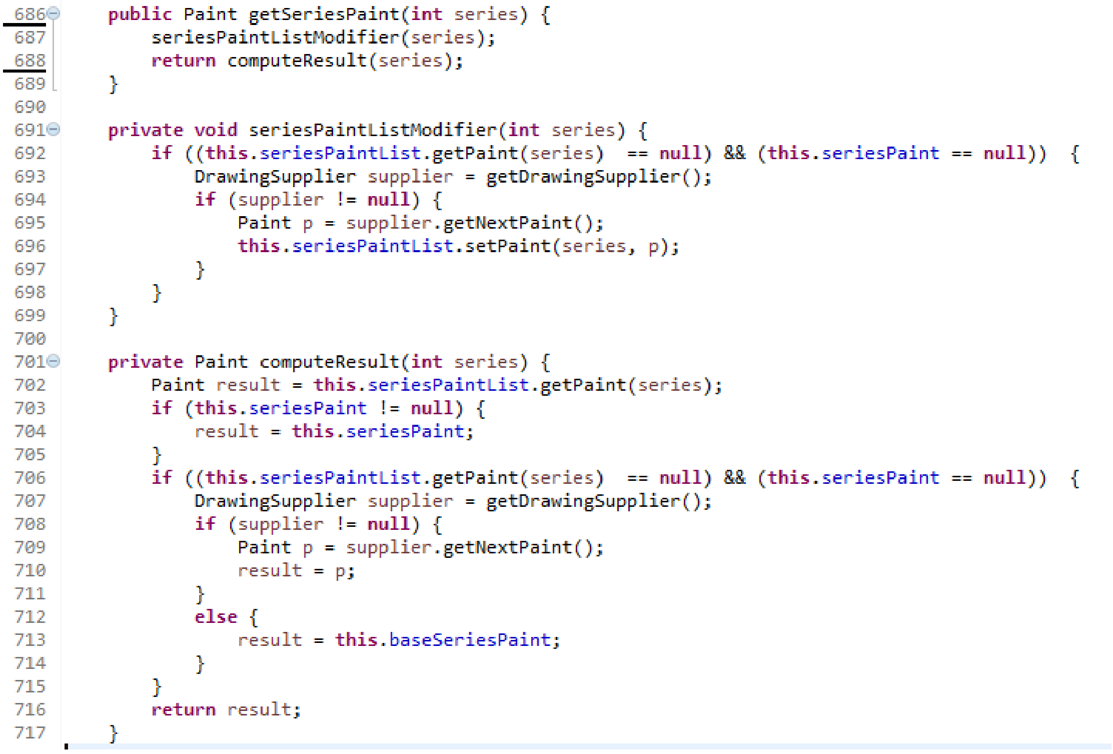
We perform inline temp on the Query.



```
public Paint getSeriesPaint(int series) {
    //Co-Slice
    if ((this.seriesPaintList.getPaint(series)  == null) && (this.seriesPaint == null))  {
        DrawingSupplier supplier = getDrawingSupplier();
        if (supplier != null) {
            Paint p = supplier.getNextPaint();
            this.seriesPaintList.setPaint(series, p);
        }
    }
    return computeResult(series);

}

private Paint computeResult(int series) {
    Paint result = this.seriesPaintList.getPaint(series);
    if (this.seriesPaint != null) {
        result = this.seriesPaint;
    }
    if ((this.seriesPaintList.getPaint(series)  == null) && (this.seriesPaint == null))  {
        DrawingSupplier supplier = getDrawingSupplier();
        if (supplier != null) {
            Paint p = supplier.getNextPaint();
            result = p;
        }
        else {
            result = this.baseSeriesPaint;
        }
    }
    return result;
}
```

Step5:

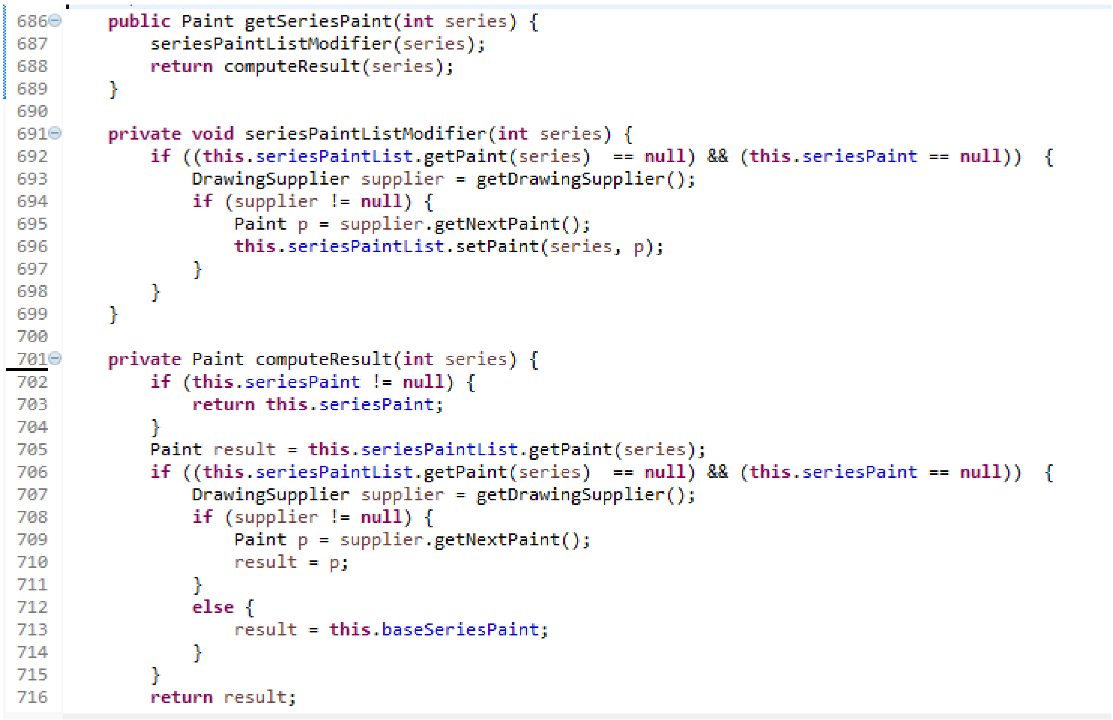
Extract method to modifier.



```
public Paint getSeriesPaint(int series) {
    seriesPaintListModifier(series);
    return computeResult(series);
}

private void seriesPaintListModifier(int series) {
    if ((this.seriesPaintList.getPaint(series)  == null) && (this.seriesPaint == null))  {
        DrawingSupplier supplier = getDrawingSupplier();
        if (supplier != null) {
            Paint p = supplier.getNextPaint();
            this.seriesPaintList.setPaint(series, p);
        }
    }
}

private Paint computeResult(int series) {
    Paint result = this.seriesPaintList.getPaint(series);
    if (this.seriesPaint != null) {
        result = this.seriesPaint;
    }
    if ((this.seriesPaintList.getPaint(series)  == null) && (this.seriesPaint == null))  {
        DrawingSupplier supplier = getDrawingSupplier();
        if (supplier != null) {
            Paint p = supplier.getNextPaint();
            result = p;
        }
        else {
            result = this.baseSeriesPaint;
        }
    }
    return result;
}
```

Step6:

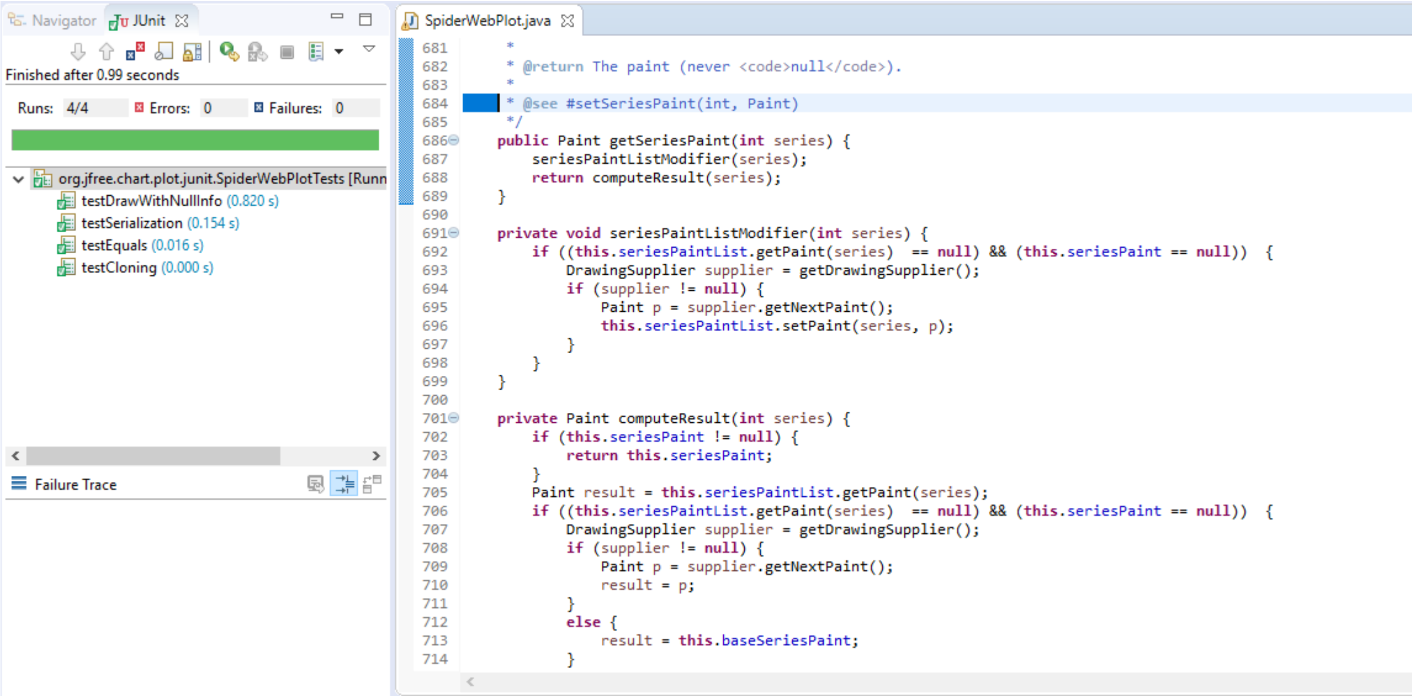
Undo to **step1** (reconstruct the return).



```
public Paint getSeriesPaint(int series) {
    seriesPaintListModifier(series);
    return computeResult(series);
}

private void seriesPaintListModifier(int series) {
    if ((this.seriesPaintList.getPaint(series)  == null) && (this.seriesPaint == null))  {
        DrawingSupplier supplier = getDrawingSupplier();
        if (supplier != null) {
            Paint p = supplier.getNextPaint();
            this.seriesPaintList.setPaint(series, p);
        }
    }
}

private Paint computeResult(int series) {
    if (this.seriesPaint != null) {
        return this.seriesPaint;
    }
    Paint result = this.seriesPaintList.getPaint(series);
    if ((this.seriesPaintList.getPaint(series)  == null) && (this.seriesPaint == null))  {
        DrawingSupplier supplier = getDrawingSupplier();
        if (supplier != null) {
            Paint p = supplier.getNextPaint();
            result = p;
        }
        else {
            result = this.baseSeriesPaint;
        }
    }
    return result;
```

Step7: no inline method needed.

The final version will be:



Finished after 0.99 seconds

Runs: 4/4 Errors: 0 Failures: 0

```
org.jfree.chart.plot.junit.SpiderWebPlotTests [Runn
    testDrawWithNullInfo (0.820 s)
    testSerialization (0.154 s)
    testEquals (0.016 s)
    testCloning (0.000 s)
```

```
     *
     * @return The paint (never <code>null</code>).
     *
     * @see #setSeriesPaint(int, Paint)
     */
    public Paint getSeriesPaint(int series) {
        seriesPaintListModifier(series);
        return computeResult(series);
    }

    private void seriesPaintListModifier(int series) {
        if ((this.seriesPaintList.getPaint(series)  == null) && (this.seriesPaint == null))  {
            DrawingSupplier supplier = getDrawingSupplier();
            if (supplier != null) {
                Paint p = supplier.getNextPaint();
                this.seriesPaintList.setPaint(series, p);
            }
        }
    }

    private Paint computeResult(int series) {
        if (this.seriesPaint != null) {
            return this.seriesPaint;
        }
        Paint result = this.seriesPaintList.getPaint(series);
        if ((this.seriesPaintList.getPaint(series)  == null) && (this.seriesPaint == null))  {
            DrawingSupplier supplier = getDrawingSupplier();
            if (supplier != null) {
                Paint p = supplier.getNextPaint();
                result = p;
            }
            else {
                result = this.baseSeriesPaint;
            }
```

Conclusions:

We explored some code-motion techniques such as sliding and bucketing and performed refactoring using those techniques. While working on the project, came to our minds how powerful and useful it can be. Even so, as programmers we found it hard to adjust the idea of preferring design improvement on efficiency, so we tried to find the balance.

After performing refactoring, mostly extracting methods, we searched for other pieces of codes that might use them as an attempt applying code eliminations, make the code shorter, and make it more efficient in some cases.

Overall, the variety of code motion and refactoring algorithms that we used in this project, could improve our code in the future to create much convenient and readable programs.