



BRUNEL UNIVERSITY LONDON
DISTRIBUTED COMPUTING SYSTEMS ENGINEERING

REPORT

EE5616

Corvin Schapöhler
Student Number: 1841781

Lab partner:
Michael Watzko

Year of Submission: 2018

Abstract

A short summary of what the project is about.

Contents

Abstract	i
1 Exercise 1	1
1.1 Method hashCode	2
1.2 Method equals	2
1.3 Method toString	2
2 Exercise 2	4
2.1 Testcases and description	4
2.1.1 Constructor	4
2.1.2 Line.add	5
2.1.3 Line.length	5
2.1.4 Line.equals	5
2.1.5 Line.hashCode	7
2.1.6 Line.toString()	8
2.1.7 Line.isValid	8
2.1.8 Line.slope	9
2.1.9 Line.intercept	10
2.2 Testresults and coverage	11
3 Exercise 3	14
3.1 Implementation and thoughts behind	14
3.2 UML class diagrams	14
3.3 UML sequence diagrams	16
3.3.1 Read lines from file	16
4 Exercise 4	18
A Appendix	19
A.1 Listings	19
A.1.1 Point.java	19
A.1.2 LineTest.java	22

A.1.3	Code for analysis	28
A.1.4	DurationTimer.java	34
A.1.5	LineClassStatistics.java	35
A.1.6	LineStatistics.java	36

Chapter 1

Exercise 1

In the first exercise a class `Point` was implemented. The developed code can be found in the Appendix (A.1). The following paragraphs will reason about different aspects of the code and why things were solved the way they were.

The class `Point` represents a point by its cartesian coordinates. For this the variables `x` and `y` were chosen. As described in the exercise the class should have two constructors.

```
1 //default ctor with x=0.0, y=0.0
2 public Point(){}
3
4 //parametricized ctor
5 public Point(double x, double y){}
```

Listing 1.1: Constructor method headers for class `Point`

Further methods for normalizing, rotating and displacing the point are given by the following methods.

```
1 //calculates distance from origin to point (normalizing
  vector)
2 public double norm(){}
3
4 //rotates point around origin by theta degrees
5 public void rotate(double theta){}
6
7 //moves the point by amount p.x and p.y
8 public void displace(Point p){}
```

Listing 1.2: Methods in class `Point`

Also the methods `hashCode`, `equals` and `toString` were overridden to coorespond to the defined behaviour. In the following section some reason is given on the specific implementation for each of these methods.

1.1 Method `hashCode`

As a hashing algorithm a very basic and simple default is provided by eclipse. This can be described by the following equation.

$$\begin{aligned} hash(p.x) &= prime \cdot 1 + (x \oplus (x \gg 32)) \\ hash(p) &= prime \cdot hash(p.x) + (y \oplus (y \gg 32)) \end{aligned}$$

Since only the values of `x` and `y` are used and no other random aspects can occur during this calculation the value for to points will be equal, if they are equal as defined in the `equals` method.

1.2 Method `equals`

The method `equals` was overridden to check the values of `x` and `y`. If those are the same `equals()` returns true, else it returns false. There are no checks for when `equals` is called with something other then another point, as this should be done by the caller before. The check `if(this == obj)` is done for faster comparison of the same object. To safely compare the values of `x` and `y`, the following code is used.

```
1  if(Double.doubleToLongBits(this.x) !=  
    Double.doubleToLongBits(other.x)) {...}
```

Listing 1.3: Safely compare double values in java

This was crucial as in Java `Double.NaN == Double.NaN` is false and nearly all error handling was done by setting the values of `x` and `y` to `NaN`.

`Double.doubleToLongBits(Double.NaN) == Double.doubleToLongBits(Double.NaN)` on the other hand is true.

1.3 Method `toString`

As the output format of `toString()` was given bei the exercise, the default `toString` method was overridden. As the commata seperator was `.` instead of `,` (which is used in germany) the output locale was set in Code.

```
1 String.format(Locale.ENGLISH, "...", x, y);
```

Listing 1.4: Setting locale to get correctly printed decimal separator

This overrides the systemwide set locale and corresponds to the correct commata separator. Also the values of x and y are printed in scientific notation with four decimals.

Chapter 2

Exercise 2

In exercise 2 the task was to implement unit tests for the class `Line`, which represents the linear regression of multiple points. In the following every test is looked at and some explanation is given on why this test was chosen. The full source code can be found at A.2.

2.1 Testcases and description

2.1.1 Constructor

As defined in the exercise two constructors should be created, a default constructor that creates an empty line with no points and a parametrized one, that initializes the line with given points. For this three test cases were written.

1. empty line with default constructor

This test calls the default constructor and checks if an empty line is created by asserting the length equals zero.

2. initialized line with points

This test calls the parametrized constructor with an array of three points and checks if the line is correctly initialized by checking if `line.lenght()` returns three.

3. empty line with empty points array

This test checks if the parametrized constructor creates an empty line if it is called with an empty array of points. This was done to validate that the internal structure can handle empty arrays and does not crash or throw exceptions.

2.1.2 Line.add

The add method should add a given point to the line. To validate the functionality two cases were checked.

1. add additional point to line

This test checks the most common use of add. It appends one point to an existing empty line and validates, if the point is added by checking if the length is incremented.

2. add null to line

This test validates an edge case, where instead of a point the value null is given as a parameter. This should not crash or throw any NullPointerExceptions and just not add any points to the line. This was again validated by checking the length of the line after adding null.

2.1.3 Line.length

The method length() should return the current length of the line as an integer. To validate this behaviour two test was created.

1. length is created with correct length

This test checks to basic functionality of length(). For this a line with three points is created, then the length method is called to check if it returns three.

2. length increments when adding points on runtime

For this test an empty line is created and a single point is added. This should increment the length of the line by one. This was done to check if length is updated when a line is edited dynamically and is not set statically on creation.

2.1.4 Line.equals

To test the equals method seven different cases were looked at. This amount comes from the different specifications, which were written in the exercise.

1. lines are not equal with different points

There are two behaviours for the equals method. Two lines, that do not have the same points should be considered as not equal, so Line.equals() should return false. For this two lines with different points were created and checked if equals returns false as defined.

2. lines with same points in different orders

As the exercise states, lines with the same points, independent of the order should be considered as equal to each other. To check this behaviour the test creates two lines, one with the points $((0,0),(0,1),(0,2))$ and one with the points $((0,2),(0,0),(0,1))$. These lines should be considered as equal to each other which was asserted.

3. object equals itself

As one would assume, the equals method should return true when called with the same object. This was tested by creating a line and checking if it equals itself. The major reason for this test was checking if the comparison may change something while evaluating the object. This should not happen and is checked by this test.

4. lines should not be equal when different with multiple points in line

This test is a bit more complicated. Two lines, that are the same length, but with different points should not be equal. Also, the equals algorithm should deal with points double in line and not confuse them as being equal. For this two lines were created, one with the points $((0,0),(0,0))$ and one with the points $((0,0),(0,1))$. These two lines should not be equal. Depending on the implementation, the fact that one point is twice in one line can break the algorithm. To validate that the used implementation is safe this test was written.

5. lines with different length should not be equal

This test generates two lines, one with length 0 and one with length 3. Those two lines should not be equal. This test was chosen to check if length is tested, which should be that case and is faster than checking every point in one line.

6. line should not be equal with null

To check if equals can cope with null as an input without throwing a `NullPointerException`, this test calls equals on a line with length 0 and compares it to null.

7. lines with points (A,A,B) and (B,B,A) should not be equal

The idea for this test came when debugging the line class. The chosen implementation did not work correctly in cases where lines l1 $((0,0),(0,0),(1,1))$ and l2 $((0,0),(1,1),(1,1))$ were compared. Even though these lines should not be equal, the equals method returned true and assumed they were equal. To check that this misbehaviour was fixed and stays fixed during further development this testcase was implemented.

2.1.5 Line.hashCode

As the exercise states, hashCode must be implemented in a suitable manner. According to the API documentation for Object.hashCode this means that the following specifications must be fulfilled¹.

- Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the equals(obj) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

To check these specifications three tests were written.

1. testTwoLinesHaveSameHashCodeSameOrder

This test check if two lines with the same points in the same order have the same hashCode. As stated in the exercise such lines should be considered equal and should produce the same hashCode (see definitions from Java Api).

2. testTwoLinesHaveSameHashCodeDifferentOrder

Lines with only the order scrambled should be considered equal. This behaviour is validated with tests in 2.1.4. According to the Java API documentation such lines should return the same hashCode. This test is pretty similar to the test before, but validates that hashCode does work with different ordered lines.

3. testHashCodeStaysSameForMultipleCalls

As stated in the Java API: " Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer". This means as long as the JVM stays the same and the object is not manipulated on a field that is used to

¹[https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode\(\)](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode())

evaluate equality of this object it **must** produce the same hashCode. This is tested by calling hashCode twice on the same line object and validating that the result stays the same. This also checks if the hashCode is calculated with no degree of randomness that is based on for example a timestamp.

2.1.6 Line.toString()

Since the output format for a line was specified in the exercise, the toString method had to be overridden. To check the correct return value three testcases were considered, as there were three possible outputs.

1. testToStringEmptyLine

The first output is for empty lines. This was not defined in the exercise, so it was assumed, that simple () should be returned. This was tested by creating a an empty line and asserting if toString would return (). This was the most sensible approach, since the toString method should return a list of all points, which are none for an empty line.

2. testToStringOnePoint

The second testcase was to check, if a line with length = 1 gives the correct return value. This was done by creating an empty line and adding one point. The main purpose of this test was to check wether the seperators between each point is printed correctly. As there is only one point, there shouldn't be any seperators before or after the point. The output is assumed to be ((Point.x, Point.y)).

3. testToStringWithThreePointsInLine

This is the most common case. In this test multiple points are used to create the line. This shows the full output for a single line, with seperators between the points and no separation for the last point. The corresponding output is build with String.format, since it is assumed that Point.toString() works as defined.

2.1.7 Line.isValid

Up until now, the line was just an Array of points, without linear regression. The following methods are all used to calculate the corresponding line represented by the given points. The first method checks if a line is valid, for this the following definitions where used.

- A line is invalid with less then or equal to one point.

- A line is invalid if the slope can not be calculated.
- A mline is invalid if the intercept can not be calculated.
- A line is valid in all other cases

To test these definitions, three tests were used

1. **testIsValidWhenZeroPointsAreStored**

As stated a line is invalid with less than one point, so it is tested if an empty line is invalid.

2. **testIsValidWhenSlopeOrInterceptCannotBeCalculated**

This test validated whether a line is invalid if its slope or intercept can not be calculated. This is done by creating a line with three points, all of which are parallel to the y-axis. Since neither the slope nor the intercept can be calculated in this case, one test validates both definitions. As this is true for all lines, this test validates the whole behaviour.

3. **testLineIsValid**

The last case tests if the line can be valid at all by initializing a line where one can calculate slope and intercept and that has more than 2 points (in this case three points with a slope of 1 and an intercept of 0).

2.1.8 Line.slope

The method slope is defined to calculate the slope of any valid line using linear regression. Also, if the slope could not be calculated the method shall throw an `RegressionFailedException`. This is tested with four tests.

1. **testReturnsCorrectSlopeForLine**

In this case it is tested, if a correct calculation is done for three points that are all on a 45 degree angled line. (maybe insert figure here / reference). The slope of this line is asserted to be 1.0

2. **testReturnsCorrectSlopeForSixPoints**

This test is used to check whether the calculation for slope is dependent on the points being in a straight line. In this case the points are one the x or y axis respectively and the slope was calculated with WolframAlpha²

²[https://www.wolframalpha.com/input/?i=linear\(\(1,0\),\(3,0\),\(5,0\),\(0,1\),\(0,3\),\(0,5\)\)](https://www.wolframalpha.com/input/?i=linear((1,0),(3,0),(5,0),(0,1),(0,3),(0,5)))

3. **testThrowsExceptionWhenSlopeNotCalculable**

This test validates if a `RegressionFailedException` is thrown, when the line is invalid or the calculation for slope fails for any kind of reason (f.e. division by 0). For this a `Line` is created where all points are parallel to the y axis. Its is not possible (at least with linear regression) to calculate the slope, so this should throw an `RegressionFailedException`. This is asserted with `assertThrows(RegressionFailedException.class, () => l1.slope())`, which can be used since the Unittests were implemented with JUnit 5.

4. **testAddPointsOtherSlope**

The last text is used to check the caching behaviour of slope. The result shall be stored after the first calculation and only changed ones the line is manipulated. For this a line is created the slope ist calculated and then points are added to the line. After that its asserted that the slope changed and is not return from the stored cache.

2.1.9 **Line.intercept**

The last method for the class `Line` is `intercept`, which calculates the y-axis intercept of a line. Like with `slope`, `intercept` can only be calculated with a valid line. To test if `intercept` handles the calculation correctly, four tests were written

1. **testInterceptReturnsCorrectValue**

The first test is the most common one. It is expected that when creating a line the intercept is calculated correctly with linear regression. This is tested by creating a line `l1` with `((0,1),(1,2))` and asserting that `intercept` returns 1.0.

2. **testInterceptReturnsCorrectValueForLargerLines**

As for `slope` a longer line was chosen where not all points are straight on the line. The intercept was calculated with WolframAlpha again. The biggest difference to most other testcases is that we assert within some `Accuracy`, which is needed as double operations in Java are not 100% accurate by the nature of the way they are handled.

3. **testInterceptThrowsExceptionWhenNotCalculable**

Like with `slope` `intercept` should throw a `RegressionFailedException` when the line is not valid or any other arithmetic problem happens. This is checked by creating a line parallel to the y-axis and asserting that `intercept` throws a `RegressionFailedException`. As in `Slope` this is done by `assertThrows(...)`.

4. **testInterceptWithNaNforValidLine**

The last test checks how well intercept and the class Line handles NaN as point values. Since multiple tests for Double.NaN was already tested in Point to not crash at any operation, this is done for intercept.

2.2 Testresults and coverage

The following screenshots show the results (see 2.1) when running the test suite and the coverage (see 2.2) produced with the selected tests. As seen the coverage is not 100%, which should be reached as best as possible. The problem comes from the implementation of intercept() (see 2.3). The method internally calls slope() which, when it fails, will throw an RegressionFailedException, which will be thrown further up the call stack to the test. After the internal slope() call, the method calculates the y-axis intercept and checks, if the calculation was successful. This is done to handle values like Double.NaN correctly. Since slope() already checks this behaviour, the test can't reach this code and can't cover it.

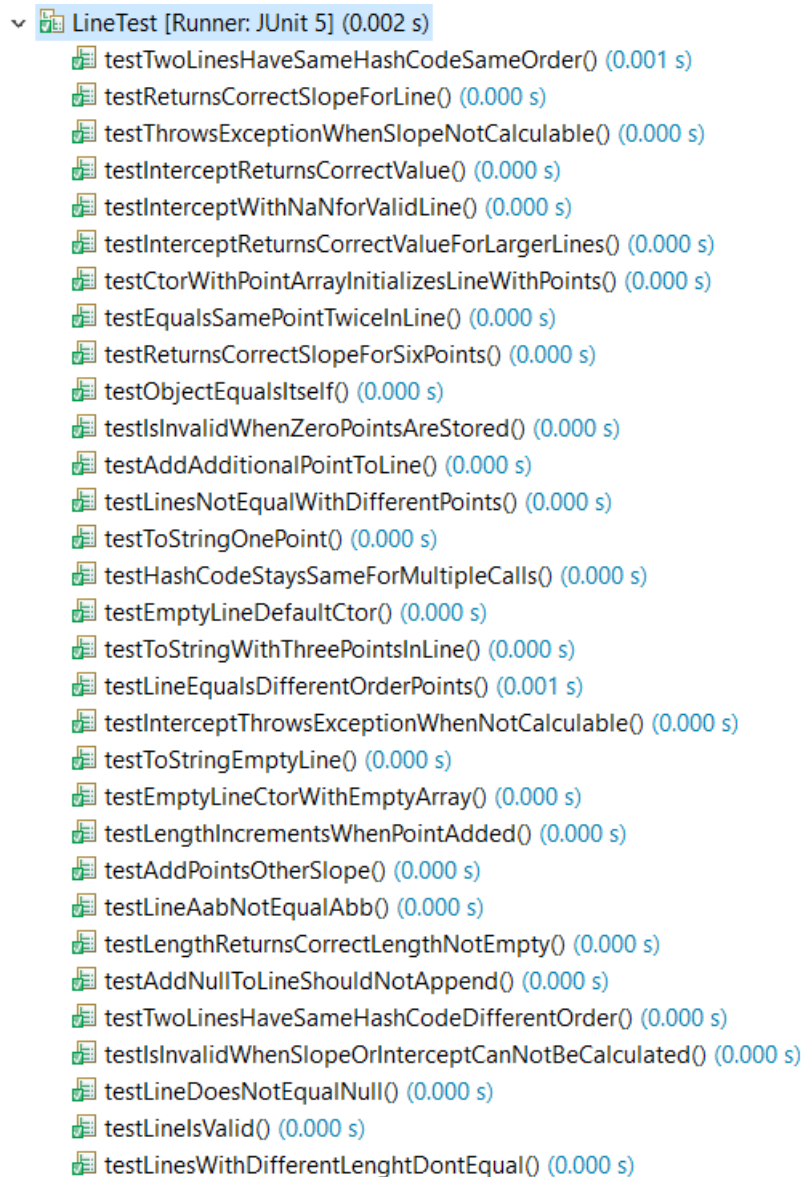


Figure 2.1: Testresults for LineTest, all tests are passed

>	LineTest.java	<div><div></div></div>	98.7 %
>	Line.java	<div><div></div></div>	98.4 %

Figure 2.2: Coverage produced by LineTest for class Line, not 100% coverage


```

    public double intercept() throws RegressionFailedException {
        if (Double.isNaN(this.intercept)) {
            this.ensureEnoughPointsForRegressionCalculation();

            double y = calcY();
            double a = slope();
            double x = calcX();

            double intercept = y - (a * x);

            if (!Double.isFinite(intercept)) {
                throw new RegressionFailedException();
            }

            this.intercept = intercept;
        }
        return this.intercept;
    }

```

Figure 2.3: Missing coverage in Line.intercept due to unreachable code

Chapter 3

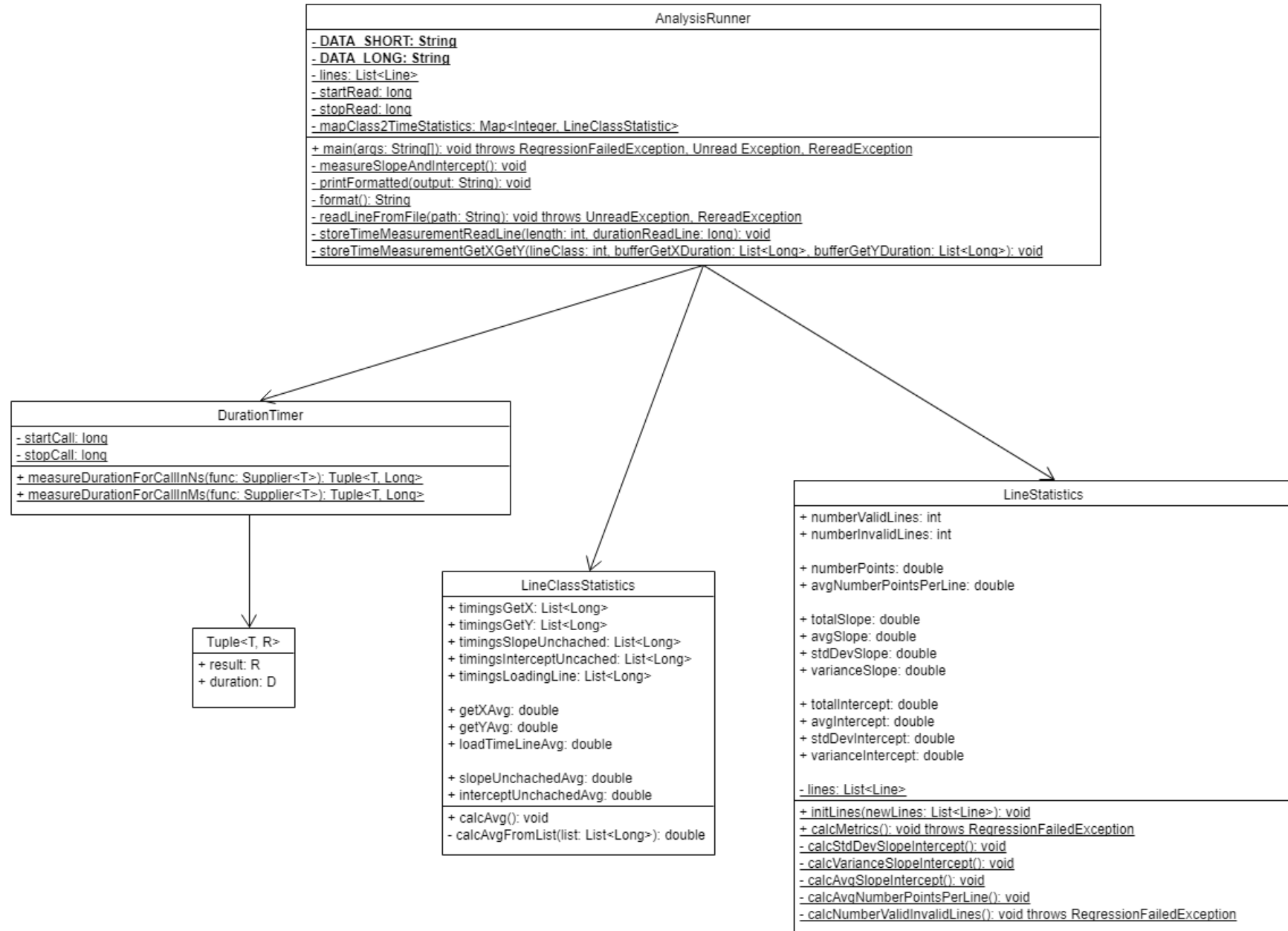
Exercise 3

3.1 Implementation and thoughts behind

In this section a detailed look into the chosen implementation.

3.2 UML class diagrams

This section shows the the UML class diagramm for measuring the data in the `data_long.dat` file. The whole code can be seen at A.1.3. For a more detailed description of the classes 3.1 is provided.



3.3 UML sequence diagrams

In this section UML sequence diagrams are provided for measuring and calculating the wanted statistics. The diagrams are splitted into methods to be more readable. In the end a single UML sequence diagram is provided that references all other provided diagrams. Also for each sequence diagram a small discription is provided and some implementation details are explained.

3.3.1 Read lines from file

The following section describes how the lines within the provided data.dat file are read and how the provided lineReader.jar is used to iterate through the file. This is especially interessting as the time needed to read in the file was measured and is analysed in 4. For this the implemented DurationTimer was used (see 3.1). It timestamps the start time, executes a given function, timestamps its ends and calculates the duration it took to execute the function. This is then returned as a tuple consisting of the result of the function and the measured duration. As seen in the provided code this is done by supplying the `Supplier<T> func` parameter, which is then called by the DurationTimer. This can also be seen in the provided sequence diagram (see 3.2).

```
1 Tuple<Double, Long> timingGetX =  
  DurationTimer.measuredurationForCallInMs(() -> reader.getX());
```

Listing 3.1: Time measurement for getX with Supplier pattern

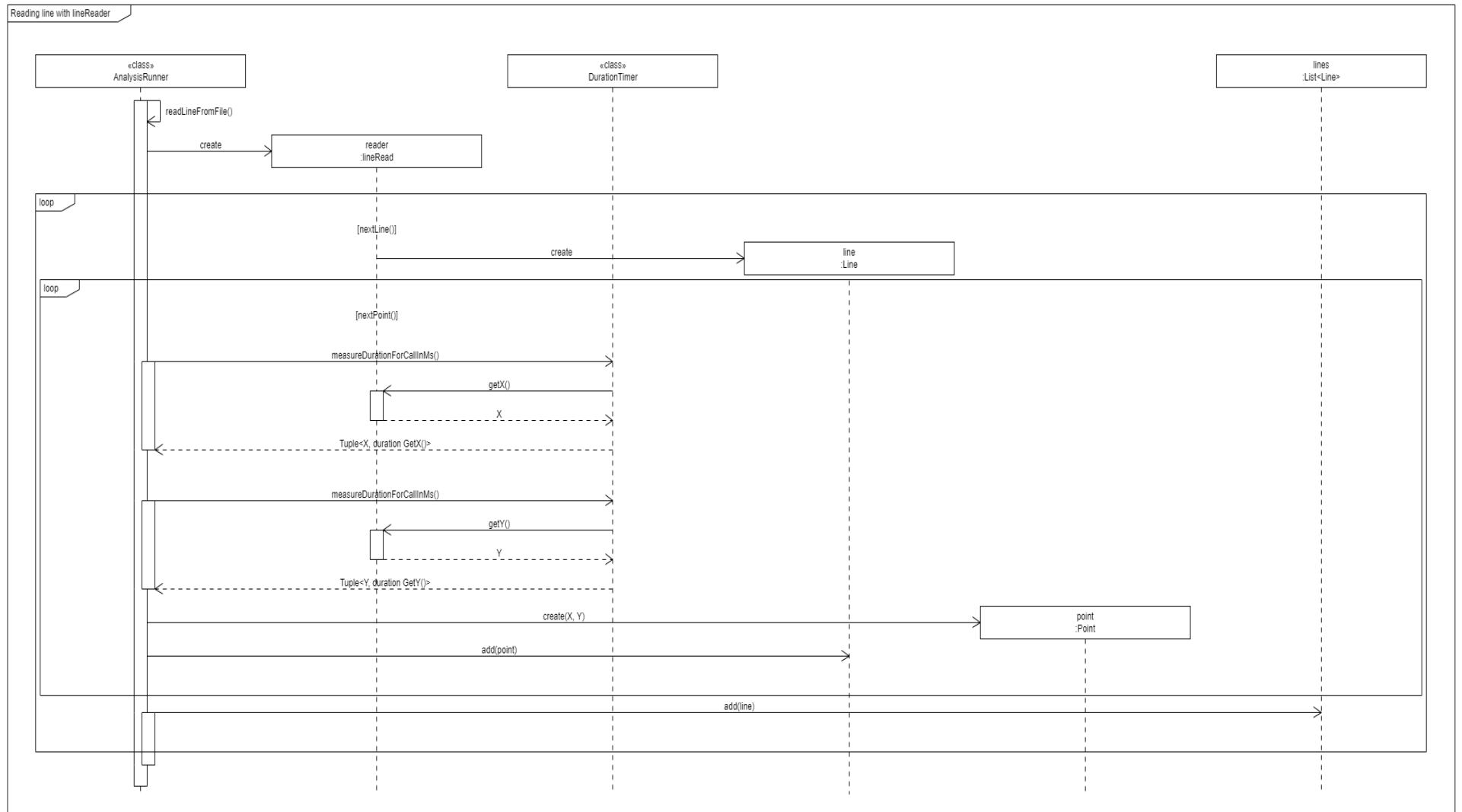


Figure 3.2: UML sequence diagram for reading in all line in provided data file

Chapter 4

Exercise 4

Investigate how the time to read in the data and perform the fit varies with the number of points in a data set. Example timing code is provided.

Appendix A

Appendix

A.1 Listings

A.1.1 Point.java

```
1 package ee5616_2018;
2
3 import java.util.Locale;
4
5 public class Point {
6
7     private double x;
8     private double y;
9
10    public Point() {
11        x = 0.0;
12        y = 0.0;
13    }
14
15    public Point(double x, double y) {
16        this.x = x;
17        this.y = y;
18    }
19
20    public double norm() {
21        return Math.sqrt((x*x) + (y*y));
22    }
23
24    public void rotate(double theta) throws
AngleOutOfRangeException {
25        if(theta < -180.0 || theta > 180.0) {
26            throw new AngleOutOfRangeException("Angle must be
between -180 and 180 degree");
```

```

27     }
28
29     double radTheta = Math.toRadians(theta);
30
31     double tempX = x * Math.cos(radTheta) - y *
Math.sin(radTheta);
32     double tempY = y * Math.cos(radTheta) + x *
Math.sin(radTheta);
33
34     x = tempX;
35     y = tempY;
36 }
37
38 public void displace(Point p) {
39     x = x + p.x;
40     y = y + p.y;
41 }
42
43 public double getX() {
44     return x;
45 }
46
47 public void setX(double x) {
48     this.x = x;
49 }
50
51 public double getY() {
52     return y;
53 }
54
55 public void setY(double y) {
56     this.y = y;
57 }
58
59 @Override
60 public int hashCode() {
61     final int prime = 31;
62     int result = 1;
63     long temp;
64     temp = Double.doubleToLongBits(x);
65     result = prime * result + (int) (temp ^ (temp >>> 32));
66     temp = Double.doubleToLongBits(y);
67     result = prime * result + (int) (temp ^ (temp >>> 32));
68     return result;
69 }
70
71 @Override

```



```

72     public boolean equals(Object obj) {
73         //Objects are same instance, faster comparison
74         if (this == obj)
75             return true;
76
77         Point other = (Point) obj;
78         if (Double.doubleToLongBits(x) !=
Double.doubleToLongBits(other.x))
79             return false;
80         if (Double.doubleToLongBits(y) !=
Double.doubleToLongBits(other.y))
81             return false;
82         return true;
83     }
84
85     @Override
86     public String toString() {
87         return String.format(Locale.ENGLISH, "( %.4E, %.4E )",
x, y);
88     }
89
90     public class AngleOutOfRangeException extends Exception{
91         private static final long serialVersionUID =
-3726276637567215315L;
92
93         public AngleOutOfRangeException(String message) {
94             super(message);
95         }
96     }
97 }

```

Listing A.1: Class Point

A.1.2 LineTest.java

```
1 package ee5616_2018;
2
3 import static org.junit.jupiter.api.Assertions.*;
4 import org.junit.jupiter.api.Test;
5
6 import ee5616_2018.Line.RegressionFailedException;
7
8 class LineTest {
9
10     public static final double ACCURACY = 0.000000000000001;
11
12     Point[] points3ordered = new Point[] {new Point(0,0), new
Point(0,1), new Point(0,2)};
13     Point[] points3scrambled = new Point[] {new Point(0,2), new
Point(0,0), new Point(0,1)};
14     Point[] points3 = new Point[] {new Point(1,0), new
Point(0,1), new Point(1,2)};
15     Point[] points45degree = new Point[] {new Point(1,1), new
Point(2,2), new Point(3,3)};
16
17
18     @Test
19     void testEmptyLineDefaultCtor() {
20         Line l = new Line();
21
22         assertEquals(0, l.length());
23     }
24
25     @Test
26     void testEmptyLineCtorWithEmptyArray() {
27         Line l = new Line(new Point[0]);
28
29         assertEquals(0, l.length());
30     }
31
32     @Test
33     void testAddAdditionalPointToLine() {
34         Line l = new Line();
35         l.add(new Point(0,1));
36
37         assertEquals(1, l.length());
38     }
39
40     @Test
```

```

41     void testAddNullToLineShouldNotAppend() {
42         Line l = new Line();
43         l.add(null);
44
45         assertEquals(0, l.length());
46     }
47
48     @Test
49     void testLengthReturnsCorrectLengthNotEmpty() {
50         Line l1 = new Line(points3);
51
52         assertEquals(3, l1.length());
53     }
54
55     @Test
56     void testLinesNotEqualWithDifferentPoints() {
57         Line l1 = new Line(points3ordered);
58         Line l2 = new Line(points3);
59
60         assertNotEquals(l1, l2);
61     }
62
63     @Test
64     void testLineEqualsDifferentOrderPoints() {
65         Line l1 = new Line(points3ordered);
66         Line l2 = new Line(points3scrambled);
67
68         assertEquals(l1, l2);
69     }
70
71     @Test
72     void testEqualsSamePointTwiceInLine() {
73         Point p1 = new Point();
74         Point p2 = new Point();
75
76         Point p3 = new Point(0,1);
77
78         Line l1 = new Line(new Point[] {p1,p2});
79         Line l2 = new Line(new Point[] {p1, p3});
80
81         assertNotEquals(l1, l2);
82     }
83
84     @Test
85     void testObjectEqualsItself() {
86         Line l1 = new Line();
87

```

```

88         assertEquals(l1, l1);
89     }
90
91     @Test
92     void testLinesWithDifferentLenghtDontEqual() {
93         Line l1 = new Line();
94         Line l2 = new Line(points3);
95
96         assertNotEquals(l1, l2);
97     }
98
99     @Test
100    void testLineDowsNotEqualNull() {
101        Line l1 = new Line();
102
103        assertNotEquals(l1, null);
104    }
105
106    @Test
107    void testTwoLinesHaveSameHashCodeDifferentOrder() {
108        Line l1 = new Line(points3ordered);
109        Line l2 = new Line(points3scrambled);
110
111        assertEquals(l1.hashCode(), l2.hashCode());
112    }
113
114    @Test
115    void testTwoLinesHaveSameHashCodeSameOrder() {
116        Line l1 = new Line(points3);
117        Line l2 = new Line(points3);
118
119        assertEquals(l1.hashCode(), l2.hashCode());
120    }
121
122    @Test
123    void testToStringEmptyLine() {
124        Line l1 = new Line();
125
126        assertEquals("()", l1.toString());
127    }
128
129    @Test
130    void testToStringOnePoint(){
131        Line l1 = new Line();
132        l1.add(new Point(0,1));
133        String wantedOutput = "(( +0.0000E+00, +1.0000E+00 ))";
134    }

```

```

135         assertEquals(wantedOutput, l1.toString());
136     }
137
138     @Test
139     void testToStringWithThreePointsInLine() {
140         Line l1 = new Line(points3);
141         String wantedOutput = String.format(
142             "(%s," + System.lineSeparator()
143             + " %s," + System.lineSeparator()
144             + " %s)", points3[0], points3[1], points3[2]);
145
146         assertEquals(wantedOutput, l1.toString());
147     }
148
149     @Test
150     void testIsInvalidWhenZeroPointsAreStored() {
151         Line l1 = new Line();
152
153         assertFalse(l1.isValid());
154     }
155
156     @Test
157     void testIsInvalidWhenOnePointIsStored() {
158         Line l1 = new Line();
159         l1.add(new Point());
160
161         assertFalse(l1.isValid());
162     }
163
164     @Test
165     void testIsInvalidWhenSlopeOrInterceptCanNotBeCalculated() {
166         Line l1 = new Line(points3ordered);
167
168         assertFalse(l1.isValid());
169     }
170
171     @Test
172     void testLineIsValid() {
173         Line l1 = new Line(points45degree);
174
175         assertTrue(l1.isValid());
176     }
177
178     @Test
179     void testReturnsCorrectSlopeForLine() throws
180     RegressionFailedException{
181         Line l1 = new Line(points45degree);

```

```

181         assertEquals(1.0, l1.slope());
182     }
183
184
185     @Test
186     void testReturnsCorrectSlopeForSixPoints() throws
187     RegressionFailedException {
188         Line l1 = new Line();
189
190         l1.add(new Point(1,0));
191         l1.add(new Point(3,0));
192         l1.add(new Point(5,0));
193         l1.add(new Point(0,1));
194         l1.add(new Point(0,3));
195         l1.add(new Point(0,5));
196
197         assertEquals(-0.627906976744186, l1.slope(), ACCURACY);
198     }
199
200     @Test
201     void testThrowsExceptionWhenSlopeNotCalculable() {
202         Line l1 = new Line(points3ordered);
203
204         assertThrows(RegressionFailedException.class, () ->
205             l1.slope());
206     }
207
208     @Test
209     void testAddPointsOtherSlope() throws
210     RegressionFailedException {
211         Line l1 = new Line(points45degree);
212
213         assertEquals(1.0, l1.slope());
214
215         l1.add(new Point(0,1));
216         l1.add(new Point(0,2));
217         l1.add(new Point(0,3));
218
219         assertEquals(1.0, l1.slope());
220     }
221
222     @Test
223     void testInterceptReturnsCorrectValue() throws
224     RegressionFailedException {
225         Point p1 = new Point(0,1);
226         Point p2 = new Point(1,2);
227         Line l1 = new Line(new Point[] {p1,p2});

```

```

224         assertEquals(1.0, l1.intercept());
225     }
226
227
228     @Test
229     void testInterceptReturnsCorrectValueForLargerLines() throws
RegressionFailedException {
230         Line l1 = new Line();
231
232         l1.add(new Point(1,0));
233         l1.add(new Point(3,0));
234         l1.add(new Point(5,0));
235         l1.add(new Point(0,1));
236         l1.add(new Point(0,3));
237         l1.add(new Point(0,5));
238
239         assertEquals(2.441860465116279, l1.intercept(),
ACCURACY);
240     }
241
242     @Test
243     void testInterceptThrowsExceptionWhenNotCalculable() {
244         Line l1 = new Line();
245
246         l1.add(new Point(0,1));
247         l1.add(new Point(0,3));
248         l1.add(new Point(0,5));
249
250         assertThrows(RegressionFailedException.class, () ->
l1.intercept());
251     }
252
253     @Test
254     void testInterceptWithNaNforValidLine() {
255         Line l1 = new Line();
256
257         l1.add(new Point(Double.NaN, Double.NaN));
258         l1.add(new Point());
259
260         assertThrows(RegressionFailedException.class, ()->
l1.intercept());
261     }
262 }

```

Listing A.2: JUnit Tests for Line

A.1.3 Code for analysis

AnalysisRunner.java

```
1 package ee5616_2018;
2
3 import java.util.ArrayList;
4 import java.util.List;
5 import java.util.Map;
6 import java.util.TreeMap;
7 import java.util.function.Supplier;
8
9 import ee5616_2018.Line.RegressionFailedException;
10 import ex3.DurationTimer;
11 import ex3.LineClassStatistic;
12 import ex3.LineStatistics;
13 import ex3.Tuple;
14 import uk.ac.brunel.ee.RereadException;
15 import uk.ac.brunel.ee.UnreadException;
16 import uk.ac.brunel.ee.lineRead;
17
18 public class AnalysisRunner {
19
20     //Paths to dat files
21     private static final String DATA_SHORT =
22         "C:\\Workspace\\TAE\\EE5616\\data_short.dat";
23     private static final String DATA_LONG =
24         "C:\\Workspace\\TAE\\EE5616\\data_long.dat";
25
26     //lines to analyse
27     private static List<Line> lines = new ArrayList<>();
28
29     //timestamp start reading and end reading for display purpose
30     private static long startRead = 0l;
31     private static long stopRead = 0l;
32
33     //Map classes to statistics for this class
34     private static Map<Integer, LineClassStatistic>
35     mapClass2TimeStatistics = new TreeMap<>();
36
37     public static void main(String[] args) throws
38         RegressionFailedException, UnreadException, RereadException {
39
40         long startExecution = System.currentTimeMillis();
41         startRead = System.currentTimeMillis();
42         readLineFromFile(DATA_SHORT);
43         stopRead = System.currentTimeMillis();
```



```

40
41     //Uncached slope and intercept calc
42     measureSlopeAndIntercept();
43
44
45     //Initialize lines in LineStatistics by reference
46     LineStatistics.initLines(lines);
47     LineStatistics.calcMetrics();
48     mapClass2TimeStatistics.forEach((lineClass,
49     lineClassStatistics) -> lineClassStatistics.calcAvgs());
50
51     long startPrint = System.currentTimeMillis();
52     printFormatted(format());
53     long endPrint = System.currentTimeMillis();
54
55     System.out.println(String.format("Printing Results took
56     %d milliseconds", endPrint - startPrint));
57
58     long stopExecution = System.currentTimeMillis();
59     System.out.println(String.format("Execution took %d
60     seconds", (stopExecution-startExecution) / 1000));
61 }
62
63 private static void measureSlopeAndIntercept() {
64     for (Line line : lines) {
65         LineClassStatistic lcs =
66         mapClass2TimeStatistics.get(line.length());
67         Tuple<Double, Long> timeSlope =
68         DurationTimer.measureDurationForCallInNs(() -> {
69             try {
70                 return line.slope();
71             } catch (RegressionFailedException e) {
72                 //Happens when line is not valid, as we
73                 cannot check without caching slope and intercept
74                 return Double.NaN;
75             }
76         });
77
78         Tuple<Double, Long> timeIntercept =
79         DurationTimer.measureDurationForCallInNs(() -> {
80             try {
81                 return line.intercept();
82             } catch (RegressionFailedException e) {
83                 //Happens when line is not valid, as we
84                 cannot check without caching slope and intercept
85                 return Double.NaN;
86             }
87         });

```

```

79         });
80
81
82         lcs.timingsSlopeUnchached.add(timeSlope.getDuration());
83
84         lcs.timingsInterceptUncached.add(timeIntercept.getDuration());
85     }
86 }
87
88 private static void printFormatted(String output) {
89     System.out.println(output);
90 }
91
92 private static String format() {
93     StringBuilder sb = new StringBuilder();
94
95     //Timing Section NYI
96     sb.append("-----"
97         + System.lineSeparator());
98     sb.append("                TIMING"
99         + System.lineSeparator());
100    sb.append("-----"
101        + System.lineSeparator());
102
103    sb.append(System.lineSeparator());
104
105    sb.append(String.format("Time needed to read file: %ds
106%n %n", (stopRead - startRead) / 1000));
107
108    sb.append("Timings per points in line (avg)" +
109        System.lineSeparator());
110    sb.append("-----" +
111        System.lineSeparator());
112    sb.append(System.lineSeparator());
113    sb.append(String.format("%6s %21s %21s %21s %21s %31s
114%n", "Class*",
115        "getX (ms)", "getY (ms)",
116        "slope (ms)", "intercept (ms)",
117        "Loadtimes for lines(ms)**"));
118    for (int i : mapClass2TimeStatistics.keySet()) {
119        LineClassStatistic lcs =
120            mapClass2TimeStatistics.get(i);
121        sb.append(String.format("%5d; %20.7f; %20.7f;
122        %20.7f; %20.7f; %30.2f; %n",
123            i, lcs.getXAvg, lcs.getYAvg,
124            lcs.slopeUnchachedAvg / (double) 1000000,

```

```

lcs.interceptUnchachedAvg/ (double) 1000000,
lcs.loadTimeLineAvg));
117     }
118     sb.append(System.lineSeparator());
119     sb.append(" *   each class represents lines with n points
(f.e. Class 2 means all lines with length=2)" +
System.lineSeparator());
120     sb.append("**   including time for measuring times for
getX and getY"+ System.lineSeparator());
121
122
123     sb.append(System.lineSeparator());
124     //Metrics Section
125     sb.append("-----"
+ System.lineSeparator());
126     sb.append("                METRICS"
+ System.lineSeparator());
127     sb.append("-----"
+ System.lineSeparator());
128     sb.append(String.format("Total number of lines: %d ( %d
valid | %d invalid ) %n",
129         LineStatistics.numberInvalidLines +
130         LineStatistics.numberValidLines ,
131         LineStatistics.numberValidLines ,
132         LineStatistics.numberInvalidLines));
133     sb.append(String.format("Average number of points
(valid) line %.2f %n", LineStatistics.avgNumberPointsPerLine
));
134     sb.append(String.format("%20s %10f %30s %10f %n",
135         "Average slope:", LineStatistics.avgSlope,
136         "standard deviation slope:",
137         LineStatistics.stdDevSlope));
138     sb.append(String.format("%20s %10f %30s %10f %n",
139         "Average intercept:",
140         LineStatistics.avgIntercept,
141         "standard deviation intercept:",
142         LineStatistics.stdDevIntercept));
143
144     return sb.toString();
145 }
146
147 private static void readLineFromFile(String path) throws
UnreadException, RereadException {
148     lineRead reader = new lineRead(path);
149
150     List<Long> bufferGetX Durations = new ArrayList<>();
151     List<Long> bufferGetY Durations = new ArrayList<>();

```

```

150
151     while (reader.nextLine()) {
152         long startTimeReadLine = System.currentTimeMillis();
153         Line line = new Line();
154         while (reader.nextPoint()) {
155             Tuple<Double, Long> getX =
DurationTimer.measureDurationForCallInMs(() -> {
156                 try {
157                     return reader.getX();
158                 } catch (RereadException e) {
159                     return Double.NaN;
160                 }
161             });
162
163             Tuple<Double, Long> getY =
DurationTimer.measureDurationForCallInMs(() -> {
164                 try {
165                     return reader.getY();
166                 } catch (RereadException e) {
167                     return Double.NaN;
168                 }
169             });
170
171             //Buffer time measurement for getX and getY
until end of line (unknown length of line until then)
172             bufferGetXDurations.add(getX.getDuration());
173             bufferGetYDurations.add(getY.getDuration());
174
175             line.add(new Point(getX.getResult(),
getY.getResult()));
176         }
177
178         long stopTimeReadLine = System.currentTimeMillis();
179         //Store results from time measurement GetX and GetY
180         storeTimeMeasurementsGetXGetY(line.length(),
bufferGetXDurations, bufferGetYDurations);
181
182         //calc duration for Line and add to Class Stats
183         storeTimeMeasurementReadLine(line.length(),
stopTimeReadLine-startTimeReadLine);
184
185         //reset buffers
186         bufferGetXDurations.clear();
187         bufferGetYDurations.clear();
188
189         lines.add(line);
190     }

```

```

191     }
192
193     private static void storeTimeMeasurementReadLine(int length,
194     long durationReadLine) {
195         //dont have to check if key is there as it is checked
196         when storing getX and getY which is called first
197         //Normally not smart, but for this it will do
198         mapClass2TimeStatistics.get(length)
199             .timingsLoadingLine.add(durationReadLine);
200     }
201
202     private static void storeTimeMeasurementsGetXGetY(int
203     lineClass, List<Long> bufferGetXDurations,
204     List<Long> bufferGetYDurations) {
205
206         if (mapClass2TimeStatistics.containsKey(lineClass)) {
207             mapClass2TimeStatistics.get(lineClass)
208                 .timingsGetX.addAll(bufferGetXDurations);
209             mapClass2TimeStatistics.get(lineClass)
210                 .timingsGetY.addAll(bufferGetYDurations);
211         } else {
212             LineClassStatistic lcs = new LineClassStatistic();
213             lcs.timingsGetX.addAll(bufferGetXDurations);
214             lcs.timingsGetY.addAll(bufferGetYDurations);
215
216             mapClass2TimeStatistics.put(lineClass, lcs);
217         }
218     }
219 }

```

Listing A.3: Starter class for analysis with main()

Tuple.java

```

1 package ex3;
2
3 public class Tuple<R,D> {
4     private R result;
5     private D duration;
6
7     public Tuple(R result, D duration) {
8         this.result = result;
9         this.duration = duration;
10    }
11 }

```

```

12     public D getDuration() {
13         return duration;
14     }
15
16     public R getResult() {
17         return result;
18     }
19
20     public void setDuration(D duration) {
21         this.duration = duration;
22     }
23
24     public void setResult(R result) {
25         this.result = result;
26     }
27 }

```

Listing A.4: TupleClass which is used to return measured time and result of function call

A.1.4 DurationTimer.java

```

1  package ex3;
2
3  import java.util.function.Supplier;
4
5  public class DurationTimer {
6      private static long startCall;
7      private static long stopCall;
8
9      public static<T> Tuple<T, Long>
10     measureDurationForCallInNs(Supplier<T> func) {
11         //Measure time in nanoseconds (stamp before and after
12         call)
13         startCall = System.nanoTime();
14         T result = func.get();
15         stopCall = System.nanoTime();
16
17         long duration = stopCall-startCall;
18
19         return new Tuple<T, Long>(result, duration);
20     }
21
22     public static <T> Tuple<T, Long>
23     measureDurationForCallInMs(Supplier<T> func) {

```

```

21         //Measure time in nanoseconds (stamp before and after
    call)
22         startCall = System.currentTimeMillis();
23         T result = func.get();
24         stopCall = System.currentTimeMillis();
25
26         long duration = stopCall - startCall;
27
28         return new Tuple<>(result, duration);
29     }
30 }

```

Listing A.5: DurationTimer to measure timings of function calls that provide any type of result

A.1.5 LineClassStatistics.java

```

1  package ex3;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class LineClassStatistic {
7      //Everything is kept public for more readable AnalysisRunner
    impl. This is normally a bad idea but not mission critical
    this time
8      public List<Long> timingsGetX = new ArrayList<>();
9      public List<Long> timingsGetY = new ArrayList<>();
10     public List<Long> timingsSlopeUnchached = new ArrayList<>();
11     public List<Long> timingsInterceptUncached = new
    ArrayList<>();
12     public List<Long> timingsLoadingLine = new ArrayList<>();
13
14     public double getXAvg = Double.NaN;
15     public double getYAvg = Double.NaN;
16     public double loadTimeLineAvg = Double.NaN;
17     public double slopeUnchachedAvg = Double.NaN;
18     public double interceptUnchachedAvg = Double.NaN;
19
20     public void calcAvgs() {
21         getXAvg = calcAvgFromList(timingsGetX);
22         getYAvg = calcAvgFromList(timingsGetY);
23         slopeUnchachedAvg =
    calcAvgFromList(timingsSlopeUnchached);

```

```

24         interceptUnchachedAvg =
        calcAvgFromList(timingsInterceptUncached);
25         loadTimeLineAvg = calcAvgFromList(timingsLoadingLine);
26     }
27
28     private double calcAvgFromList(List<Long> list) {
29         return list.stream().mapToLong(a ->
        a).average().orElseGet(() -> Double.NaN);
30     }
31 }

```

Listing A.6: LineClassStatistics which hold all timings statistics corresponding to one single line class

A.1.6 LineStatistics.java

```

1  package ex3;
2
3  import java.util.List;
4
5  import ee5616_2018.Line;
6  import ee5616_2018.Line.RegressionFailedException;
7
8  public class LineStatistics {
9      //Everything public as in LineClassStatistics, this is only
        for easier access
10     public static int numberValidLines = 0;
11     public static int numberInvalidLines = 0;
12
13     public static double numberPoints = 0.0;
14     public static double avgNumberPointsPerLine = Double.NaN;
15
16     public static double totalSlope = 0.0;
17     public static double avgSlope = Double.NaN;
18     public static double stdDevSlope = Double.NaN;
19     public static double varianceSlope = 0.0;
20
21     public static double totalIntercept = 0.0;
22     public static double avgIntercept = Double.NaN;
23     public static double stdDevIntercept = Double.NaN;
24     public static double varianceIntercept = 0.0;
25
26     private static List<Line> lines = null;
27
28     public static void initLines(List<Line> newLines) {

```



```

29         lines = newLines;
30     }
31
32     public static void calcMetrics() throws
RegressionFailedException {
33
34         //Check if lines is initialized, else return with no
calculation
35         if(lines == null) {
36             return;
37         }
38
39         //1. number of Valid Lines and Invalid Lines
40         //1a. count all points
41         //1b. For Valid Lines calc slope
42         calcNumberVaildInvalidLines();
43         //2. Avg Number of Points per Line
44         calcAvgNumberPointsPerLine();
45         //3. Avg for slope() and intercept()
46         calcAvgSlopeIntercept();
47         //5. calc Variance slope and intercept (avg needed for
this)
48         calcVarianceSlopeIntercept();
49         //6. calc std-dev slope and intercept
50         calcStdDevSlopeIntercept();
51     }
52     private static void calcStdDevSlopeIntercept() {
53         stdDevIntercept = Math.sqrt(varianceIntercept);
54         stdDevSlope = Math.sqrt(varianceSlope);
55     }
56     private static void calcVarianceSlopeIntercept() throws
RegressionFailedException {
57         for (Line line : lines) {
58             //if not valid, slope and intercept not calculable,
skip this line
59             if (!line.isValid()) continue;
60
61             //Variance = ((current - avg)^2) / count items
62             varianceIntercept += ((line.intercept() -
avgIntercept) * (line.intercept() - avgIntercept))/ (double)
numberValidLines;
63             varianceSlope += ((line.slope() - avgSlope) *
(line.slope() - avgSlope))/ (double) numberValidLines;
64         }
65     }
66     private static void calcAvgSlopeIntercept() {

```

```

67         avgSlope = (double) totalSlope / (double)
numberValidLines;
68         avgIntercept = (double) totalIntercept / (double)
numberValidLines;
69     }
70     private static void calcAvgNumberPointsPerLine() {
71         //cast to double to get correct result, else it always
would cut off floating points
72         avgNumberPointsPerLine = (double) numberPoints /
(double) numberValidLines;
73     }
74     private static void calcNumberVaildInvalidLines() throws
RegressionFailedException {
75         for (Line l : lines) {
76             if (l.isValid()) {
77                 numberPoints += l.length();
78                 numberValidLines++;
79                 totalIntercept += l.intercept();
80                 totalSlope += l.slope();
81             } else {
82                 numberInvalidLines++;
83             }
84         }
85     }
86 }

```

Listing A.7: LineStatistics holds all statistics correspondign to the whole set of Lines