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 Analysis and validation 14

3.2 UML class diagrams 14

3.3 UML sequence diagrams 16

3.3.1 Read lines from file 16

3.3.2 Measure slope and intercept 18

3.3.3 Calculate averages for line class statistics 20

4 Exercise 4 **22**

A Appendix **23**

A.1 Listings 23

A.1.1 Point.java 23
A.1.2 LineTest.java 26
A.1.3 Code for analysis 32
A.1.4 DurationTimer.java 39
A.1.5 LineClassStatistics.java 40
A.1.6 LineStatistics.java 41
A.2 Raw data collected 44

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 coodinates. For this the variables x and y were choosen. As described in the exercise the class should have two constructors.

```
    //default ctor with x=0.0, y=0.0
    public Point(){}

    //parametrized ctor
    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.

```
    //calculates distance from origin to point (normalizing
   vector)
    public double norm(){}

    //rotates point around origin by theta degrees
    public void rotate(double theta){}

    //moves the point by amount p.x and p.y
    public void displace(Point p){}
```

Listing 1.2: Methods in class Point

Also the methods hashcode, equals and toString were overriden to cooorespond 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.

$$hash(p.x) = prime \cdot 1 + (x \oplus (x \gg 32))$$

$$hash(p) = prime \cdot hash(p.x) + (y \oplus (y \gg 32))$$

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.

```
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.

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

Listing 1.4: Setting locale to get correctly printed decimal seperator

This overrides the systemwide set locale and corresponds to the correct commata seperator. 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 choosen. 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 contructors 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.lentgh() 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 beeing equal. For this two lines where 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 choosen to check if length is tested, which should be that case and is faster then 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 choosen 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 this 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 fulfilled[1].

- 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()`

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 returnm 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 seperation 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 where used

1. **testIsInvalidWhenZeroPointsAreStored**
   As stated a line is invalid with less then one point, so it is tested if an empty line is invalid.

2. **testIsInvalidWhenSlopeOrInterceptCanNotBeCalculated**
   This test validated wether 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-axsis. 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 then 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 calulation is done for three points that are all on a 45 degree angled line. (maybe instert figutre here / reference). The slope of this line is asserted to be 1.0

2. **testReturnsCorrectSlopeForSixPoints**
   This test is used to check wetehr the calculation for slope is dependend on the points beeing in a straight line. In this case the points are one the x or y axis respectively and the slope was calculated with WolframAlpha[2]

[2] `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 where 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 where 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 choosen 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 opertations 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 form 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 sucessful. This is done to handle values like Double.NaN correclty. Since slope() already checks this behaviour, the test can't reach this code and can't cover it.

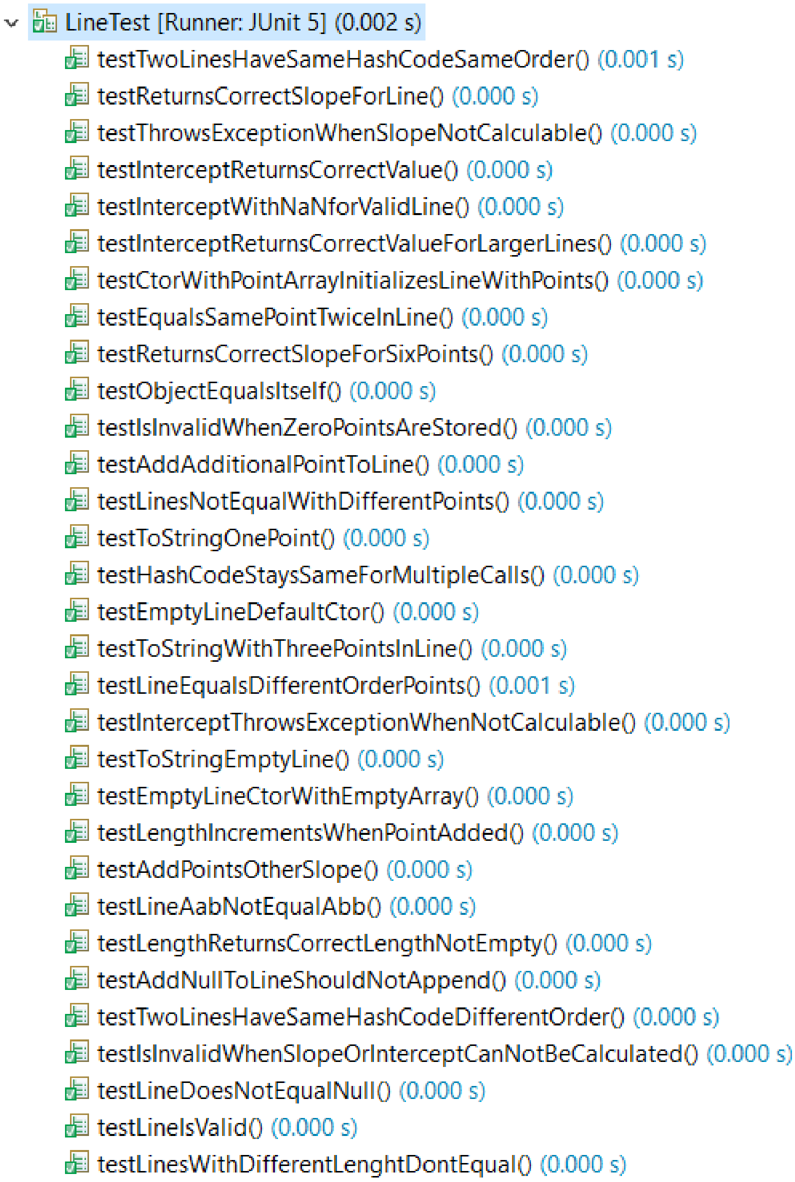


Figure 2.1: Testresults for LineTest, all tests are passed

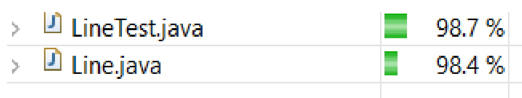


Figure 2.2: Coverage produced by LineTest for class Line, not 100% coverage

```java
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

This chapter takes a closer look at the AnalysisRunner. Especially some validation is given on why it's assumed that the chosen imlementation works as specified.

3.1 Analysis and validation

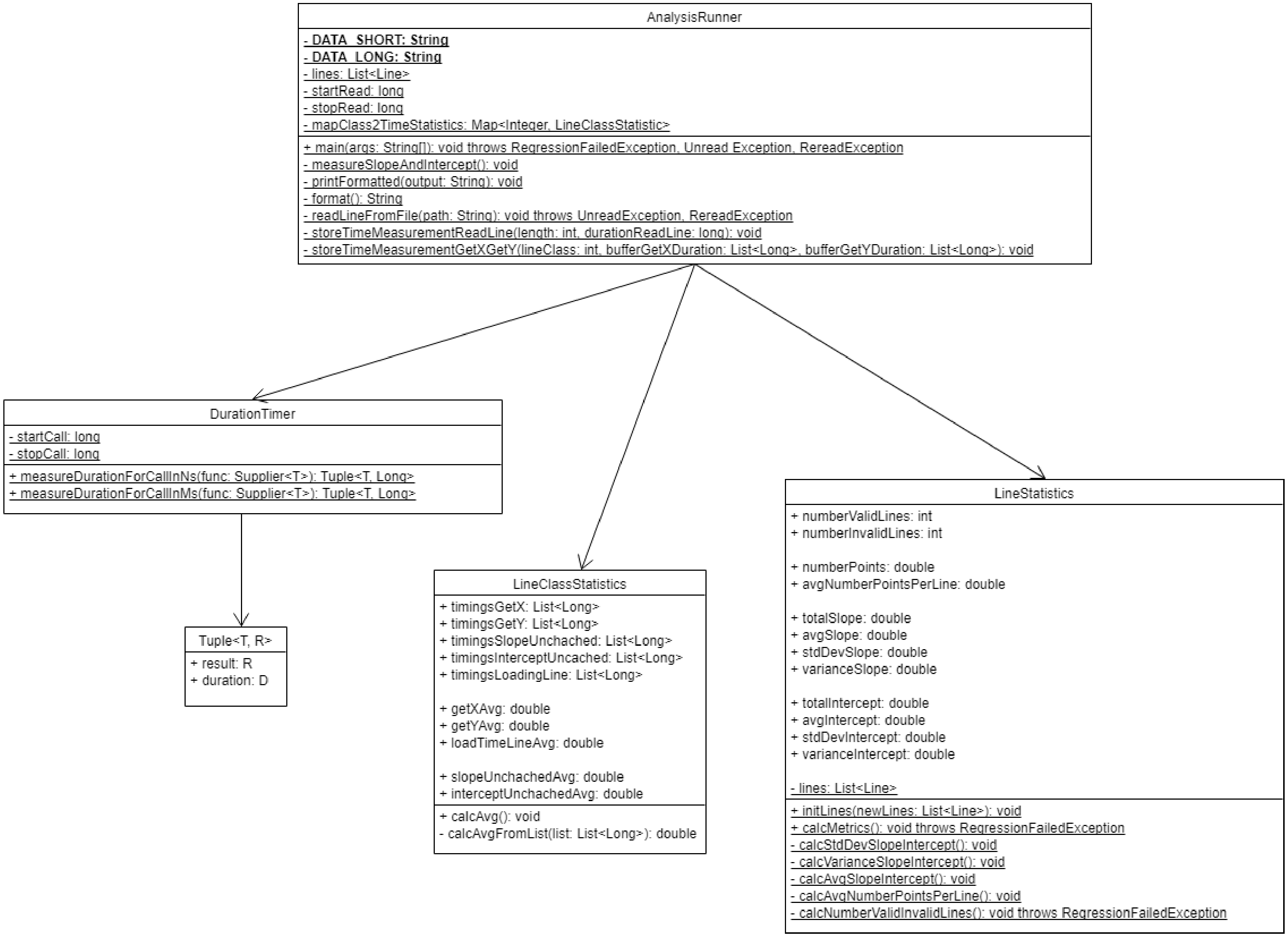
To check wheter the chosen implementation works as specified some sanity checks were used.

- Validate results for data_short.dat
- Validate results by comparison with other lab participants
- Test coverage for Line and Point classes

The sanity checks resulted in the assumption that the implementation is correct. All asked lab participants had close results for data_short and data_long. The differences might come from JVM optimizations or calculations with double types, which in Java is not 100% accurate.

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.


AnalysisRunner
- DATA_SHORT: String
- DATA_LONG: String
- lines: List<Line>
- startRead: long
- stopRead: long
- mapClass2TimeStatistics: Map<Integer, LineClassStatistic>
+ main(args: String[]): void throws RegressionFailedException, Unread Exception, RereadException
- measureSlopeAndIntercept(): void
- printFormatted(output: String): void
- format(): String
- readLineFromFile(path: String): void throws UnreadException, RereadException
- storeTimeMeasurementReadLine(length: int, durationReadLine: long): void
- storeTimeMeasurementGetXGetY(lineClass: int, bufferGetXDuration: List<Long>, bufferGetYDuration: List<Long>): void
DurationTimer
- startCall: long
- stopCall: long
+ measureDurationForCallInNs(func: Supplier<T>): Tuple<T, Long>
+ measureDurationForCallInMs(func: Supplier<T>): Tuple<T, Long>
Tuple<T, R>
+ result: R
+ duration: D
LineClassStatistics
+ timingsGetX: List<Long>
+ timingsGetY: List<Long>
+ timingsSlopeUnchached: List<Long>
+ timingsInterceptUncached: List<Long>
+ timingsLoadingLine: List<Long>
+ getXAvg: double
+ getYAvg: double
+ loadTimeLineAvg: double
+ slopeUnchachedAvg: double
+ interceptUnchachedAvg: double
+ calcAvg(): void
- calcAvgFromList(list: List<Long>): double
LineStatistics
+ numberValidLines: int
+ numberInvalidLines: int
+ numberPoints: double
+ avgNumberPointsPerLine: double
+ totalSlope: double
+ avgSlope: double
+ stdDevSlope: double
+ varianceSlope: double
+ totalIntercept: double
+ avgIntercept: double
+ stdDevIntercept: double
+ varianceIntercept: double
- lines: List<Line>
+ initLines(newLines: List<Line>): void
+ calcMetrics(): void throws RegressionFailedException
- calcStdDevSlopeIntercept(): void
- calcVarianceSlopeIntercept(): void
- calcAvgSlopeIntercept(): void
- calcAvgNumberPointsPerLine(): void
- calcNumberValidInvalidLines(): void throws RegressionFailedException

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

This section describes how the provided datafiles are read and the created lines are created. The provided sequence diagram 3.2 shows how the dataflow is done while reading each line. Firstly, within AnalysisRunner the main method is called, which calls the static method AnalysisRunner.readLinesFromFile(). This creates an object of the provided lineReader class called reader and initializes the path to `private static final String DATA_LONG`. Then the initialized reader object is used to iterate over the whole data set. This is done via `reader.nextLine()` which retuns true when there are still lines to read. For each line the reader is used to iterate over every point in the line. For each point the `getX()` and `getY()` methods are used to get the X and Y coordinate. This process is timestamped and the duration is calculated. For this `DurationTimer.measureDurationForCallInMs` is used (see 3.1). This retuns a Tuple with the result of `getX()` and the time it took to read the data. This data is analysed in 4

```
 Tuple<Double, Long> timingGetX =
DurationTimer.measureDurationForCallInMs(() -> reader.getX());
 //Some Code here
 Tuple<Double, Long> timingGetY =
DurationTimer.measureDurationForCallInMs(() -> reader.getY());
```

Listing 3.1: Measure duration for calls getX

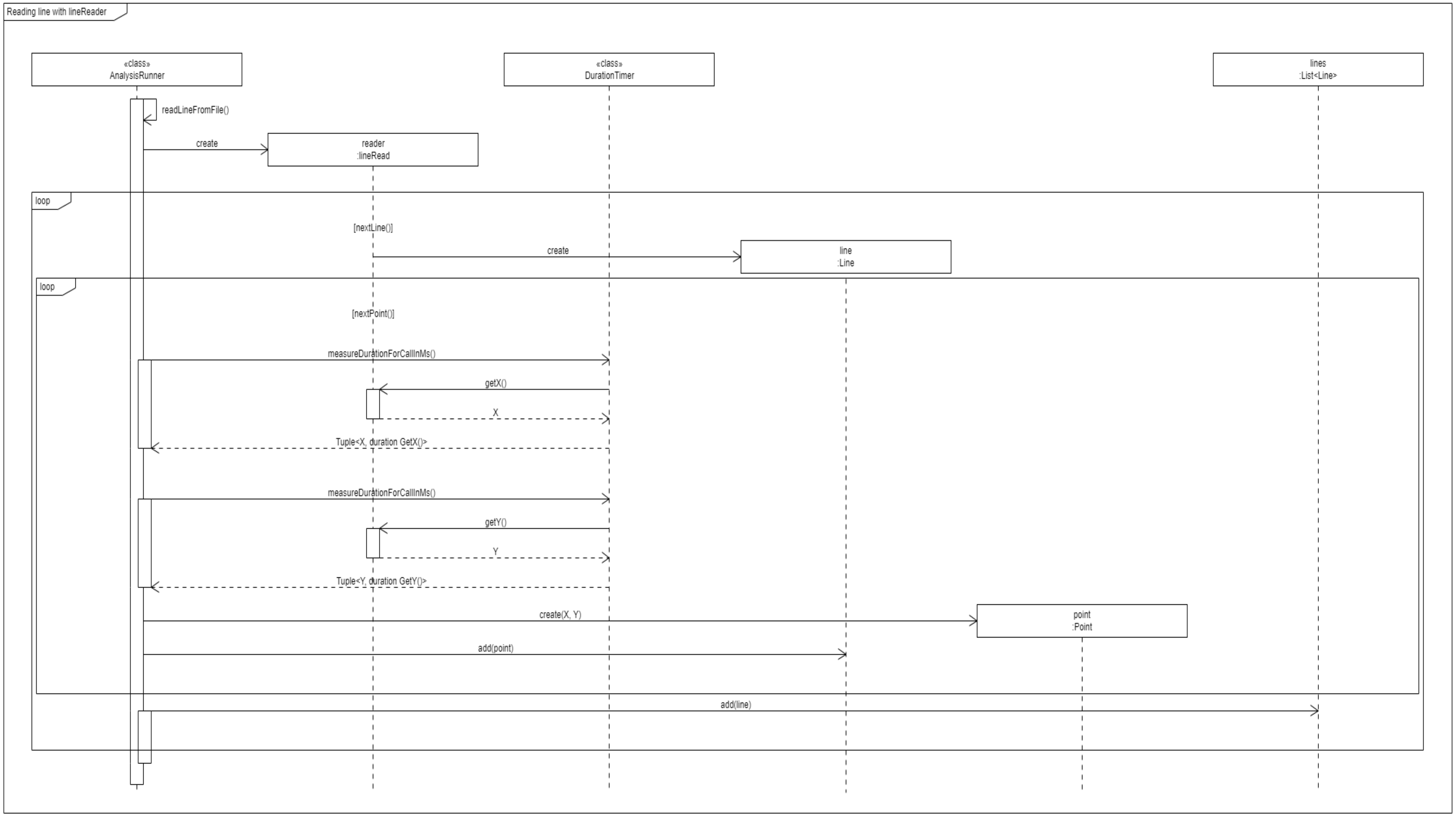


Figure 3.2: UML sequence diagram for reading in all line in provided data file

3.3.2 Measure slope and intercept

This method is called before any other method that would calculate slope or intercept, especially `line.isValid()`, as those methods would calculate slope and intercept and therefore cache the result. To prevent this from happening, slope is also called before intercept else the same problem would occur. The provided sequence diagram 3.4 shows the dataflow and how the DurationTimer is again used to measure the needed time for calculating slope and intercept. It has to be noted, that for measuring the slope and intercept uncached some error handling has to be done in `DurationTimer.measureDurationForCallInNs()` (see 3.2).

```
Tuple<Double, Long> timeSlope = DurationTimer.measureDurationForCallInNs(()-> {
    try {
        return line.slope();
    } catch (RegressionFailedException e) {
        //Happens when line is not valid, as we cannot check without caching slope and intercept
        return Double.NaN;
    }
});
```

Listing 3.2: Error handling in supplier for unchaced slope and intercept

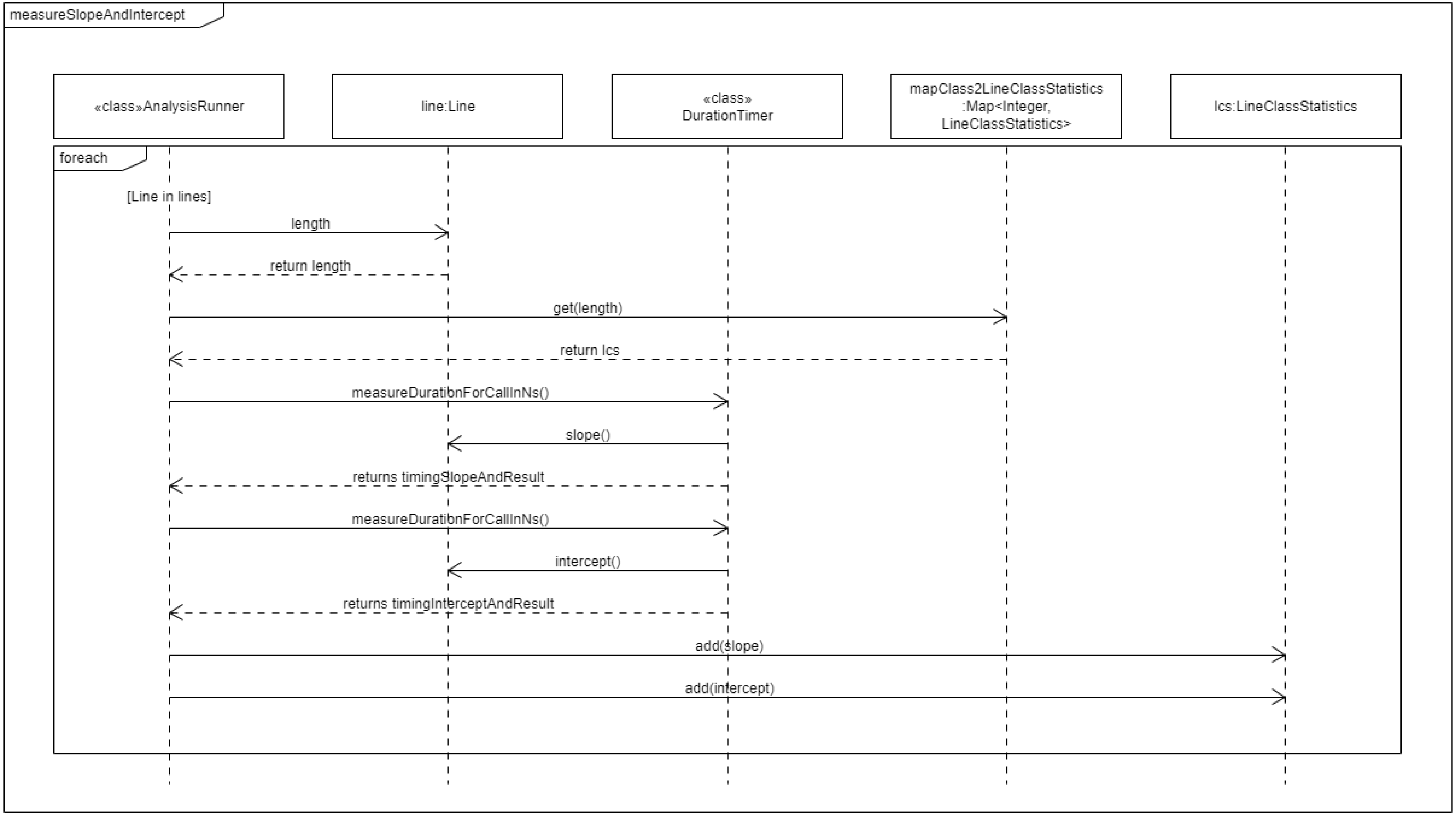


Figure 3.3: UML sequence diagram for calculating and measuring time and intercept

3.3.3 Calculate averages for line class statistics

The last sequence diagram shows the calculations of all averages for any line class. All the collected statistics are timeings of some sort they are all stored as lists of long values. The average is calculated with Java streams (see 3.3).

```java
private double calcAvgFromList(List<Long> list) {
    return list.stream().mapToLong(a ->
   a).average().orElseGet(() -> Double.NaN);
}
```

Listing 3.3: Java stream used to calculate average of a list

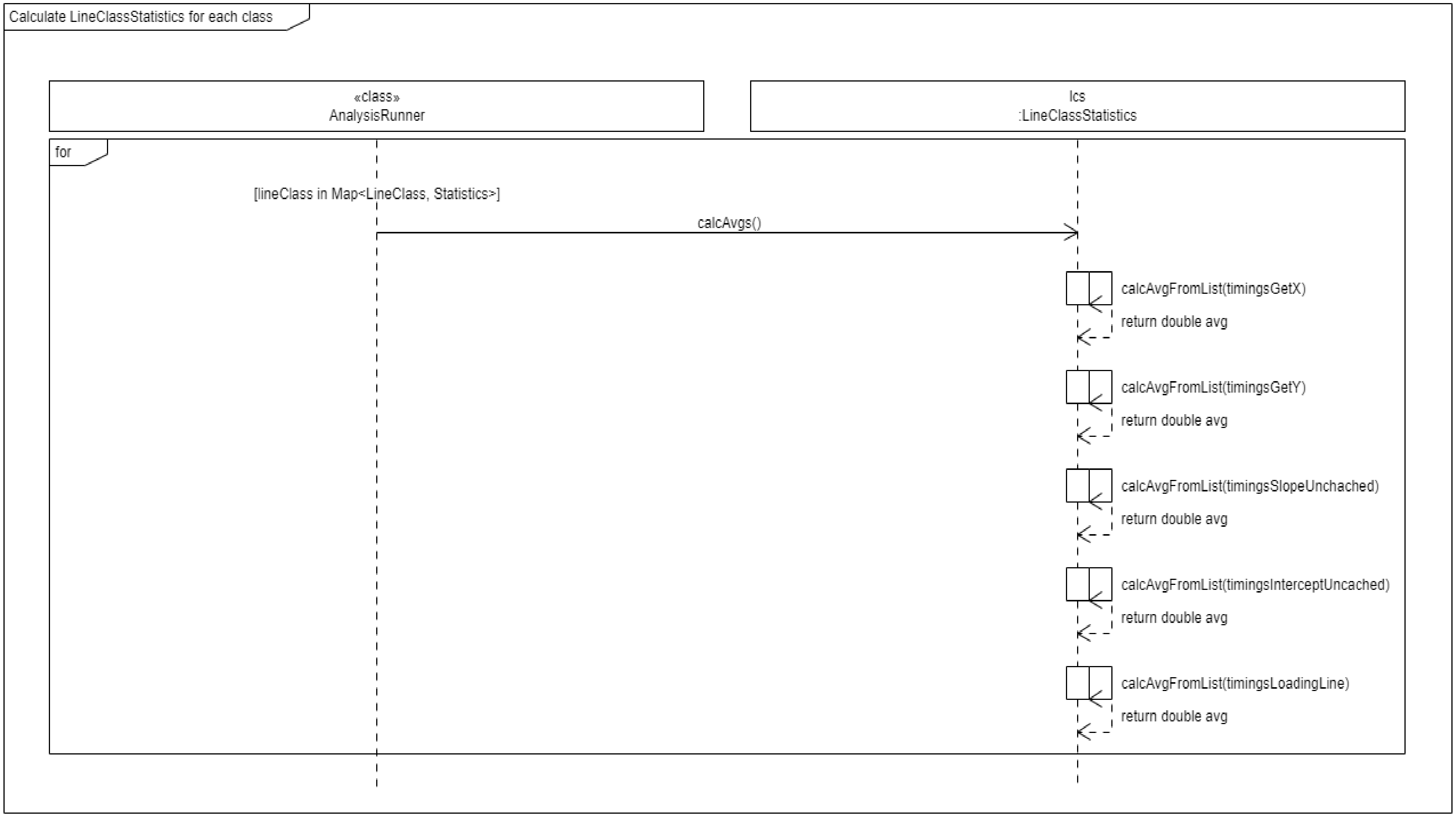


Figure 3.4: UML sequence diagram describing the calculation of all averages for a line class

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.

The topic of this chapter is the evaluation ofthe collected data from Chapter 3. For this the output shown in **??**

Appendix A

Appendix

A.1 Listings

A.1.1 Point.java

```java
package ee5616_2018;

import java.util.Locale;

public class Point {

    private double x;
    private double y;

    public Point() {
        x = 0.0;
        y = 0.0;
    }

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double norm() {
        return Math.sqrt((x*x) + (y*y));
    }

    public void rotate(double theta) throws
  AngleOutOfRangeException {
        if(theta < -180.0 || theta > 180.0) {
            throw new AngleOutOfRangeException("Angle must be
  between -180 and 180 degree");
```

```java
        }

        double radTheta = Math.toRadians(theta);

        double tempX = x * Math.cos(radTheta) - y *
    Math.sin(radTheta);
        double tempY = y * Math.cos(radTheta) + x *
    Math.sin(radTheta);

        x = tempX;
        y = tempY;
    }

    public void displace(Point p) {
        x = x + p.x;
        y = y + p.y;
    }

    public double getX() {
        return x;
    }

    public void setX(double x) {
        this.x = x;
    }

    public double getY() {
        return y;
    }

    public void setY(double y) {
        this.y = y;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        long temp;
        temp = Double.doubleToLongBits(x);
        result = prime * result + (int) (temp ^ (temp >>> 32));
        temp = Double.doubleToLongBits(y);
        result = prime * result + (int) (temp ^ (temp >>> 32));
        return result;
    }

    @Override
```

```java
    public boolean equals(Object obj) {
        //Objects are same instance, faster comparison
        if (this == obj)
            return true;

        Point other = (Point) obj;
        if (Double.doubleToLongBits(x) != Double.doubleToLongBits(other.x))
            return false;
        if (Double.doubleToLongBits(y) != Double.doubleToLongBits(other.y))
            return false;
        return true;
    }

    @Override
    public String toString() {
        return String.format(Locale.ENGLISH, "( %+.4E, %+.4E )", x, y);
    }

    public class AngleOutOfRangeException extends Exception{
        private static final long serialVersionUID = -3726276637567215315L;

        public AngleOutOfRangeException(String message) {
            super(message);
        }
    }
}
```

Listing A.1: Class Point

A.1.2 LineTest.java

```
package ee5616_2018;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

import ee5616_2018.Line.RegressionFailedException;

class LineTest {

    public static final double ACCURACY = 0.0000000000001;

    Point[] points3ordered = new Point[] {new Point(0,0), new
   Point(0,1), new Point(0,2)};
    Point[] points3scrambled = new Point[] {new Point(0,2), new
   Point(0,0), new Point(0,1)};
    Point[] points3 = new Point[] {new Point(1,0), new
   Point(0,1), new Point(1,2)};
    Point[] points45degree = new Point[] {new Point(1,1), new
   Point(2,2), new Point(3,3)};


    @Test
    void testEmptyLineDefaultCtor() {
        Line l = new Line();

        assertEquals(0, l.length());
    }

    @Test
    void testEmptyLineCtorWithEmptyArray() {
        Line l = new Line(new Point[0]);

        assertEquals(0, l.length());
    }

    @Test
    void testAddAdditionalPointToLine() {
        Line l = new Line();
        l.add(new Point(0,1));

        assertEquals(1, l.length());
    }

    @Test
```

```java
    void testAddNullToLineShouldNotAppend() {
        Line l = new Line();
        l.add(null);

        assertEquals(0,l.length());
    }

    @Test
    void testLengthReturnsCorrectLengthNotEmpty() {
        Line l1 = new Line(points3);

        assertEquals(3, l1.length());
    }

    @Test
    void testLinesNotEqualWithDifferentPoints() {
        Line l1 = new Line(points3ordered);
        Line l2 = new Line(points3);

        assertNotEquals(l1, l2);
    }

    @Test
    void testLineEqualsDifferentOrderPoints() {
        Line l1 = new Line(points3ordered);
        Line l2 = new Line(points3scrambled);

        assertEquals(l1, l2);
    }

    @Test
    void testEqualsSamePointTwiceInLine() {
        Point p1 = new Point();
        Point p2 = new Point();

        Point p3 = new Point(0,1);

        Line l1 = new Line(new Point[] {p1,p2});
        Line l2 = new Line(new Point[] {p1, p3});

        assertNotEquals(l1, l2);
    }

    @Test
    void testObjectEqualsItself() {
        Line l1 = new Line();

```

```
        assertEquals(l1, l1);
    }

    @Test
    void testLinesWithDifferentLenghtDontEqual() {
        Line l1 = new Line();
        Line l2 = new Line(points3);

        assertNotEquals(l1, l2);
    }

    @Test
    void testLineDowsNotEqualNull() {
        Line l1 = new Line();

        assertNotEquals(l1, null);
    }

    @Test
    void testTwoLinesHaveSameHashCodeDifferentOrder() {
        Line l1 = new Line(points3ordered);
        Line l2 = new Line(points3scrambled);

        assertEquals(l1.hashCode(), l2.hashCode());
    }

    @Test
    void testTwoLinesHaveSameHashCodeSameOrder() {
        Line l1 = new Line(points3);
        Line l2 = new Line(points3);

        assertEquals(l1.hashCode(), l2.hashCode());
    }

    @Test
    void testToStringEmptyLine() {
        Line l1 = new Line();

        assertEquals("()", l1.toString());
    }

    @Test
    void testToStringOnePoint(){
        Line l1 = new Line();
        l1.add(new Point(0,1));
        String wantedOutput = "(( +0.0000E+00, +1.0000E+00 ))";

```

```
        assertEquals(wantedOutput, l1.toString());
    }

    @Test
    void testToStringWithThreePointsInLine() {
        Line l1 = new Line(points3);
        String wantedOutput = String.format(
                "(%s," + System.lineSeparator()
                + " %s," + System.lineSeparator()
                + " %s)", points3[0], points3[1], points3[2]);

        assertEquals(wantedOutput, l1.toString());
    }

    @Test
    void testIsInvalidWhenZeroPointsAreStored() {
        Line l1 = new Line();

        assertFalse(l1.isValid());
    }

    @Test
    void testIsInvalidWhenOnePointIsStored() {
        Line l1 = new Line();
        l1.add(new Point());

        assertFalse(l1.isValid());
    }

    @Test
    void testIsInvalidWhenSlopeOrInterceptCanNotBeCalculated() {
        Line l1 = new Line(points3ordered);

        assertFalse(l1.isValid());
    }

    @Test
    void testLineIsValid() {
        Line l1 = new Line(points45degree);

        assertTrue(l1.isValid());
    }

    @Test
    void testReturnsCorrectSlopeForLine() throws
   RegressionFailedException{
        Line l1 = new Line(points45degree);
```

```

        assertEquals(1.0, l1.slope());
    }

    @Test
    void testReturnsCorrectSlopeForSixPoints() throws RegressionFailedException {
        Line l1 = new Line();

        l1.add(new Point(1,0));
        l1.add(new Point(3,0));
        l1.add(new Point(5,0));
        l1.add(new Point(0,1));
        l1.add(new Point(0,3));
        l1.add(new Point(0,5));

        assertEquals(-0.627906976744186, l1.slope(), ACCURACY);
    }

    @Test
    void testThrowsExceptionWhenSlopeNotCalculable() {
        Line l1 = new Line(points3ordered);

        assertThrows(RegressionFailedException.class, () -> l1.slope());
    }

    @Test
    void testAddPointsOtherSlope() throws RegressionFailedException {
        Line l1 = new Line(points45degree);

        assertEquals(1.0, l1.slope());

        l1.add(new Point(0,1));
        l1.add(new Point(0,2));
        l1.add(new Point(0,3));

        assertNotEquals(1.0, l1.slope());
    }

    @Test
    void testInterceptReturnsCorrectValue() throws RegressionFailedException {
        Point p1 = new Point(0,1);
        Point p2 = new Point(1,2);
        Line l1 = new Line(new Point[] {p1,p2});
```

```java

        assertEquals(1.0, l1.intercept());
    }

    @Test
    void testInterceptReturnsCorrectValueForLargerLines() throws
    RegressionFailedException {
        Line l1 = new Line();

        l1.add(new Point(1,0));
        l1.add(new Point(3,0));
        l1.add(new Point(5,0));
        l1.add(new Point(0,1));
        l1.add(new Point(0,3));
        l1.add(new Point(0,5));

        assertEquals(2.441860465116279, l1.intercept(),
    ACCURACY);
    }

    @Test
    void testInterceptThrowsExceptionWhenNotCalculable() {
        Line l1 = new Line();

        l1.add(new Point(0,1));
        l1.add(new Point(0,3));
        l1.add(new Point(0,5));

        assertThrows(RegressionFailedException.class, () ->
    l1.intercept());
    }

    @Test
    void testInterceptWithNaNforValidLine() {
        Line l1 = new Line();

        l1.add(new Point(Double.NaN, Double.NaN));
        l1.add(new Point());

        assertThrows(RegressionFailedException.class, ()->
    l1.intercept());
    }
}
```

Listing A.2: JUnit Tests for Line

A.1.3 Code for analysis

AnalysisRunner.java

```java
package ee5616_2018;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.TreeMap;
import java.util.function.Supplier;

import ee5616_2018.Line.RegressionFailedException;
import ex3.DurationTimer;
import ex3.LineClassStatistic;
import ex3.LineStatistics;
import ex3.Tuple;
import uk.ac.brunel.ee.RereadException;
import uk.ac.brunel.ee.UnreadException;
import uk.ac.brunel.ee.lineRead;

public class AnalysisRunner {

    //Paths to dat files
    private static final String DATA_SHORT =
    "C:\\Workspace\\TAE\\EE5616\\data_short.dat";
    private static final String DATA_LONG =
    "C:\\Workspace\\TAE\\EE5616\\data_long.dat";

    //lines to analyse
    private static List<Line> lines = new ArrayList<>();

    //timestamp start reading and end reading for display purpose
    private static long startRead = 0l;
    private static long stopRead = 0l;

    //Map classes to statistics for this class
    private static Map<Integer, LineClassStatistic>
    mapClass2TimeStatistics = new TreeMap<>();

    public static void main(String[] args) throws
    RegressionFailedException, UnreadException, RereadException {

        long startExecution = System.currentTimeMillis();
        startRead = System.currentTimeMillis();
        readLineFromFile(DATA_SHORT);
        stopRead = System.currentTimeMillis();
```

```

        //Uncached slope and intercept calc
        measureSlopeAndIntercept();


        //Initialize lines in LineStatistics by reference
        LineStatistics.initLines(lines);
        LineStatistics.calcMetrics();
        mapClass2TimeStatistics.forEach((lineClass,
    lineClassStatistics) -> lineClassStatistics.calcAvgs());

        long startPrint = System.currentTimeMillis();
        printFormatted(format());
        long endPrint = System.currentTimeMillis();

        System.out.println(String.format("Printing Results took
    %d miliseconds", endPrint - startPrint));

        long stopExecution = System.currentTimeMillis();
        System.out.println(String.format("Execution took %d
    seconds", (stopExecution-startExecution) / 1000));
    }

    private static void measureSlopeAndIntercept() {
        for (Line line : lines) {
            LineClassStatistic lcs =
    mapClass2TimeStatistics.get(line.length());
            Tuple<Double, Long> timeSlope =
    DurationTimer.measureDurationForCallInNs(()-> {
                try {
                    return line.slope();
                } catch (RegressionFailedException e) {
                    //Happens when line is not valid, as we
    cannot check without caching slope and intercept
                    return Double.NaN;
                }
            });

            Tuple<Double, Long> timeIntercept =
    DurationTimer.measureDurationForCallInNs(()-> {
                try {
                    return line.intercept();
                } catch (RegressionFailedException e) {
                    //Happens when line is not valid, as we
    cannot check without caching slope and intercept
                    return Double.NaN;
                }
```

```java
            });


    lcs.timingsSlopeUnchached.add(timeSlope.getDuration());

    lcs.timingsInterceptUncached.add(timeIntercept.getDuration());

        }
    }

    private static void printFormatted(String output) {
        System.out.println(output);
    }

    private static String format() {
        StringBuilder sb = new StringBuilder();

        //Timing Section NYI
        sb.append("--------------------------------------------"
            + System.lineSeparator());
        sb.append("                   TIMING"
            + System.lineSeparator());
        sb.append("--------------------------------------------"
            + System.lineSeparator());

        sb.append(System.lineSeparator());

        sb.append(String.format("Time needed to read file: %ds
%n %n", (stopRead - startRead) / 1000));

        sb.append("Timings per points in line (avg)" +
System.lineSeparator());
        sb.append("________________________________" +
System.lineSeparator());
        sb.append(System.lineSeparator());
        sb.append(String.format("%6s %21s %21s %21s %21s %31s
%n", "Class*",
                "getX (ms)", "getY (ms)",
                "slope (ms)", "intercept (ms)",
                "Loadtimes for lines(ms)**"));
        for (int i : mapClass2TimeStatistics.keySet()) {
            LineClassStatistic lcs =
mapClass2TimeStatistics.get(i);
            sb.append(String.format("%5d; %20.7f; %20.7f;
%20.7f; %20.7f; %30.2f; %n",
                    i, lcs.getXAvg, lcs.getYAvg,
lcs.slopeUnchachedAvg / (double) 1000000,
```

```
lcs.interceptUnchachedAvg/ (double) 1000000,
lcs.loadTimeLineAvg));
        }
        sb.append(System.lineSeparator());
        sb.append(" *  each class represents lines with n points
(f.e. Class 2 means all lines with length=2)" +
System.lineSeparator());
        sb.append("**  including time for measuring times for
getX and getY"+ System.lineSeparator());


        sb.append(System.lineSeparator());
        //Metrics Section
        sb.append("--------------------------------------------"
            + System.lineSeparator());
        sb.append("                   METRICS"
            + System.lineSeparator());
        sb.append("--------------------------------------------"
            + System.lineSeparator());
        sb.append(String.format("Total number of lines: %d ( %d
valid | %d invalid ) %n",
                LineStatistics.numberInvalidLines +
LineStatistics.numberValidLines,
                LineStatistics.numberValidLines,
LineStatistics.numberInvalidLines));
        sb.append(String.format("Average number of points
(valid) line %.2f %n", LineStatistics.avgNumberPointsPerLine
));
        sb.append(String.format("%20s %10f %30s %10f %n",
                "Average slope:", LineStatistics.avgSlope,
                "standard deviation slope:",
LineStatistics.stdDevSlope));
        sb.append(String.format("%20s %10f %30s %10f %n",
                "Average intercept:",
LineStatistics.avgIntercept,
                "standard deviation intercept:",
LineStatistics.stdDevIntercept));

        return sb.toString();
    }

    private static void readLineFromFile(String path) throws
UnreadException, RereadException {
        lineRead reader = new lineRead(path);

        List<Long> bufferGetXDurations = new ArrayList<>();
        List<Long> bufferGetYDurations = new ArrayList<>();
```

```java

        while (reader.nextLine()) {
            long startTimeReadLine = System.currentTimeMillis();
            Line line = new Line();
            while (reader.nextPoint()) {
                Tuple<Double, Long> getX = DurationTimer.measureDurationForCallInMs(() -> {
                    try {
                        return reader.getX();
                    } catch (RereadException e) {
                        return Double.NaN;
                    }
                });

                Tuple<Double, Long> getY = DurationTimer.measureDurationForCallInMs(() -> {
                    try {
                        return reader.getY();
                    } catch (RereadException e) {
                        return Double.NaN;
                    }
                });

                //Buffer time measurement for getX and getY until end of line (unknown length of line until then)
                bufferGetXDurations.add(getX.getDuration());
                bufferGetYDurations.add(getY.getDuration());

                line.add(new Point(getX.getResult(), getY.getResult()));
            }

            long stopTimeReadLine = System.currentTimeMillis();
            //Store results from time measurement GetX and GetY
            storeTimeMeasurementsGetXGetY(line.length(), bufferGetXDurations, bufferGetYDurations);

            //calc duration for Line and add to Class Stats
            storeTimeMeasurementReadLine(line.length(), stopTimeReadLine-startTimeReadLine);

            //reset buffers
            bufferGetXDurations.clear();
            bufferGetYDurations.clear();

            lines.add(line);
        }
```

```java
    }

    private static void storeTimeMeasurementReadLine(int length, long durationReadLine) {
        //dont have to check if key is there as it is checked when storing getX and getY which is called first
        //Normally not smart, but for this it will do
        mapClass2TimeStatistics.get(length)
            .timingsLoadingLine.add(durationReadLine);
    }

    private static void storeTimeMeasurementsGetXGetY(int lineClass, List<Long> bufferGetXDurations,
            List<Long> bufferGetYDurations) {

        if (mapClass2TimeStatistics.containsKey(lineClass)) {
            mapClass2TimeStatistics.get(lineClass)
                .timingsGetX.addAll(bufferGetXDurations);
            mapClass2TimeStatistics.get(lineClass)
                .timingsGetY.addAll(bufferGetYDurations);
        } else {
            LineClassStatistic lcs = new LineClassStatistic();
            lcs.timingsGetX.addAll(bufferGetXDurations);
            lcs.timingsGetY.addAll(bufferGetYDurations);

            mapClass2TimeStatistics.put(lineClass, lcs);
        }

    }
}
```

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

Tuple.java

```java
package ex3;

public class Tuple<R,D> {
    private R result;
    private D duration;

    public Tuple(R result, D duration) {
        this.result = result;
        this.duration = duration;
    }

    public D getDuration() {
        return duration;
    }

    public R getResult() {
        return result;
    }

    public void setDuration(D duration) {
        this.duration = duration;
    }

    public void setResult(R result) {
        this.result = result;
    }
}
```

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

A.1.4 DurationTimer.java

```java
package ex3;

import java.util.function.Supplier;

public class DurationTimer {
    private static long startCall;
    private static long stopCall;

    public static<T> Tuple<T, Long> measureDurationForCallInNs(Supplier<T> func) {
        //Measure time in nanoseconds (stamp before and after call)
        startCall = System.nanoTime();
        T result = func.get();
        stopCall = System.nanoTime();

        long duration =  stopCall-startCall;

        return new Tuple<T, Long>(result, duration);
    }

    public static <T> Tuple<T, Long> measureDurationForCallInMs(Supplier<T> func) {
        //Measure time in nanoseconds (stamp before and after call)
        startCall = System.currentTimeMillis();
        T result = func.get();
        stopCall = System.currentTimeMillis();

        long duration =  stopCall-startCall;

        return new Tuple<>(result, duration);
    }
}
```

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

A.1.5 LineClassStatistics.java

```java
package ex3;

import java.util.ArrayList;
import java.util.List;

public class LineClassStatistic {
    //Everything is kept public for more readable AnalysisRunner impl. This is normally a bad idea but not mission critical this time
    public List<Long> timingsGetX = new ArrayList<>();
    public List<Long> timingsGetY = new ArrayList<>();
    public List<Long> timingsSlopeUnchached = new ArrayList<>();
    public List<Long> timingsInterceptUncached = new ArrayList<>();
    public List<Long> timingsLoadingLine = new ArrayList<>();

    public double getXAvg = Double.NaN;
    public double getYAvg = Double.NaN;
    public double loadTimeLineAvg = Double.NaN;
    public double slopeUnchachedAvg = Double.NaN;
    public double interceptUnchachedAvg = Double.NaN;

    public void calcAvgs() {
        getXAvg = calcAvgFromList(timingsGetX);
        getYAvg = calcAvgFromList(timingsGetY);
        slopeUnchachedAvg = calcAvgFromList(timingsSlopeUnchached);
        interceptUnchachedAvg = calcAvgFromList(timingsInterceptUncached);
        loadTimeLineAvg = calcAvgFromList(timingsLoadingLine);
    }

    private double calcAvgFromList(List<Long> list) {
        return list.stream().mapToLong(a -> a).average().orElseGet(() -> Double.NaN);
    }
}
```

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

A.1.6 LineStatistics.java

```java
package ex3;

import java.util.List;

import ee5616_2018.Line;
import ee5616_2018.Line.RegressionFailedException;

public class LineStatistics {
    //Everything public as in LineClassStatistics, this is only for easier access
    public static int numberValidLines = 0;
    public static int numberInvalidLines = 0;

    public static double numberPoints = 0.0;
    public static double avgNumberPointsPerLine = Double.NaN;

    public static double totalSlope = 0.0;
    public static double avgSlope = Double.NaN;
    public static double stdDevSlope = Double.NaN;
    public static double varianceSlope = 0.0;

    public static double totalIntercept = 0.0;
    public static double avgIntercept = Double.NaN;
    public static double stdDevIntercept = Double.NaN;
    public static double varianceIntercept = 0.0;

    private static List<Line> lines = null;

    public static void initLines(List<Line> newLines) {
        lines = newLines;
    }

    public static void calcMetrics() throws RegressionFailedException {

        //Check if lines is initialized, else return with no calculation
        if(lines == null) {
            return;
        }

        //1. number of Valid Lines and Invalid Lines
        //1a. count all points
        //1b. For Valid Lines calc slope
```

```java
        calcNumberVaildInvalidLines();
        //2. Avg Number of Points per Line
        calcAvgNumberPointsPerLine();
        //3. Avg for slope() and intercept()
        calcAvgSlopeIntercept();
        //5. calc Variance slope and intercept (avg needed for this)
        calcVarianceSlopeIntercept();
        //6. calc std-dev slope and intercept
        calcStdDevSlopeIntercept();
    }
    private static void calcStdDevSlopeIntercept() {
        stdDevIntercept = Math.sqrt(varianceIntercept);
        stdDevSlope = Math.sqrt(varianceSlope);
    }
    private static void calcVarianceSlopeIntercept() throws RegressionFailedException {
        for (Line line : lines) {
            //if not valid, slope and intercept not calculable, skip this line
            if (!line.isValid()) continue;

            //Variance = ((current - avg)^2) / count items
            varianceIntercept += ((line.intercept() - avgIntercept) * (line.intercept() - avgIntercept))/ (double) numberValidLines;
            varianceSlope += ((line.slope() - avgSlope) * (line.slope() - avgSlope))/ (double) numberValidLines;
        }
    }
    private static void calcAvgSlopeIntercept() {
        avgSlope = (double) totalSlope / (double) numberValidLines;
        avgIntercept = (double) totalIntercept / (double) numberValidLines;
    }
    private static void calcAvgNumberPointsPerLine() {
        //cast to double to get correct result, else it always would cut off floating points
        avgNumberPointsPerLine = (double) numberPoints / (double) numberValidLines;
    }
    private static void calcNumberVaildInvalidLines() throws RegressionFailedException {
        for (Line l : lines) {
            if (l.isValid()) {
                numberPoints += l.length();
```

```
                numberValidLines++;
                totalIntercept += l.intercept();
                totalSlope += l.slope();
            } else {
                numberInvalidLines++;
            }
        }
    }
}
```

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

A.2 Raw data collected

```
-----------------------------------------------------------------
                              TIMING
-----------------------------------------------------------------

Time needed to read file: 503s

Timings per class (points  in line) (avg)

------------------------------------------

           getX       getY       slope      intercept    Loadtimes
  Class*   (ms)       (ms)       (ms)         (ms)       lines(ms)**
    1;  1.5459184;  1.6020408;  0.0085094;  0.0047119;      3.15;
    2;  1.5537500;  1.6000000;  0.0011742;  0.0006956;      6.31;
    3;  1.5440806;  1.5717884;  0.0012346;  0.0005630;      9.35;
    4;  1.5579019;  1.5619891;  0.0012083;  0.0005356;     12.48;
    5;  1.5593315;  1.5721448;  0.0021393;  0.0008533;     15.66;
    6;  1.5720507;  1.5696754;  0.0018745;  0.0007700;     18.85;
    7;  1.5798894;  1.5730556;  0.0022753;  0.0008581;     22.07;
    8;  1.5803571;  1.5859788;  0.0023656;  0.0009995;     25.33;
    9;  1.5638554;  1.5866131;  0.0025246;  0.0008673;     28.37;
   10;  1.4818898;  2.4884514;  0.0028532;  0.0008337;     39.71;
   11;  1.4700240;  2.4892086;  0.0032995;  0.0011080;     43.56;
   12;  1.4691667;  2.4854167;  0.0044326;  0.0014906;     47.47;
   13;  1.4665762;  2.4861461;  0.0030203;  0.0007844;     51.39;
   14;  1.4638313;  2.4725554;  0.0035678;  0.0012137;     55.12;
   15;  1.4585956;  2.4874899;  0.0046295;  0.0014613;     59.20;
   16;  1.4754902;  2.4852941;  0.0044011;  0.0019476;     63.52;
   17;  1.4759487;  2.4844125;  0.0043210;  0.0013800;     67.35;
   18;  1.4693732;  2.4854701;  0.0052550;  0.0014606;     71.20;
   19;  1.4691449;  2.4842351;  0.0069823;  0.0023223;     75.14;
   20;  1.4680707;  2.4889946;  0.0058923;  0.0017381;     79.15;
   21;  1.4606597;  2.4787575;  0.0064285;  0.0019345;     82.75;
   22;  1.4718615;  2.4881522;  0.0064151;  0.0021935;     87.12;
   23;  1.4730099;  2.4860323;  0.0062514;  0.0017643;     91.08;
   24;  1.4676245;  2.4872605;  0.0056327;  0.0014604;     94.95;
   25;  1.4677000;  2.4797000;  0.0055549;  0.0015799;     98.70;
```

```
 *  each class represents lines with n points
    (f.e. Class 2 means all lines with length=2)

**  including time for measuring times for getX and getY

------------------------------------------------------------------
                              METRICS
------------------------------------------------------------------
Total number of lines: 10000 ( 9608 valid | 392 invalid )
Average number of points (valid) line 13.57
Average slope:                    0.750046
standard deviation slope:         0.004738
Average intercept:               -1.920046
standard deviation intercept:     0.008197

Printing results took 28 miliseconds
Execution took 503 seconds
```