



College of Engineering, Design and Physical Science
Electronic and Computer Engineering

Assignment Java Testing and Measuring

Distributed Computing Systems Engineering Msc

Author: Christoph Gschrey

Lab-Partner: Matthias Gebert

Date: 22. September 2017

Supervisor: Prof. Dr. Peter Väterlein

ASSIGNMENT SUBMISSION FORM

Please note: that no course work will be accepted without this cover sheet.

Please ensure: that you keep a copy of work submitted and retain your receipt in case of query.

Student Number:	SPO ID Number (Office use only):	
Course:		Level:

MODULE	
Module Code:	Module Title:
Lab / Assignment:	Deadline:
Lab group (if applicable):	Date Stamp (Office use only):
Academic Responsible:	
Administrator:	

Please note: that detailed feedback will be provided on a feedback form.

✂.....

RECEIPT SECTION (Office Copy)	
Student Number:	SPO ID Number (Office use only):
Student First Name:	Student Last Name:
Module Code:	Module Title:
Lab / Assignment:	
Lab group (if applicable):	Deadline:
Academic Responsible:	Number of Days late:

DECLARATION	
I have read and I understand the guidelines on plagiarism and cheating in the Handbook and I certify that my contribution to this report fully complies with these guidelines. I confirm that I have kept a copy of my work and that I have not lent my work to any other students.	
Signed:	Date Stamp (Office use only):

✂.....

RECEIPT SECTION (Student Copy)	
Student Number:	Student Name:
Lab / Assignment:	
Lab group (if applicable):	Module Title:
Academic Responsible:	Deadline:
Module Code:	Date Stamp (Office use only):

The University penalty system will be applied to any work submitted late.

IMPORTANT: You **MUST** keep this receipt in a safe place as you may be asked to produce it at any time as proof of submission of the assignment. Please submit this form with the assignment attached to the Department of Design Education Office in the Michael Sterling Building, room MCST 055.

Contents

1	Introduction	1
2	Test Driven Development	2
3	The Point() class	3
3.1	The default constructor	3
3.2	An alternate constructor	3
3.3	getters and setters	4
3.4	The equals() method	4
3.5	The hashCode() method	5
3.6	The toString() method	6
3.7	The toString() method	6
3.8	The rotate() method	6
3.9	The displace() method	7
4	Test for the Line() class	10
4.1	Constructor tests	10
4.2	Tests for length() and add()	11
4.3	Tests for equals()	12
4.4	Tests for hashCode()	13
4.5	Tests for toString()	14
4.6	Tests for isValid()	14
4.7	Tests for slope()	15
4.8	Tests for intercept()	16
5	The LinearRegression Application	20
6	Evaluating the performance	24
	Bibliography	28
A	Appendix	i
A.0.1	Point class listing	ii
A.0.2	LineTest class listing	v
A.0.3	LinearRegressionProgram class listing	xi
A.0.4	Line Reader listing	xv

A.0.5	Excel Writer listing	xvii
-------	--------------------------------	------

1 Introduction

In the last few decades software has become an important factor in everyone's lives. New and exciting developments in this sector can be observed every day by those who are interested. This ranges from simple applications used for smartphones which are basically used by all people around the globe to more abstract and complex things like self driving cars or automated houses. The goal is to make all people's lives more comfortable. Work that is too dangerous or too exhausting is done mostly by machines these days. Although this is a development which is greatly appreciated by most people, it brings some problems as well. As software is developed by people, it can never be fully reliable. People make mistakes and this transfers to the programs they write.

This leads to the question how a better reliability and stability can be achieved for our programs. The answer of course is **testing**.

There are a lot of different methods, guides, processes and tutorials on this topic. These cannot be all listed here in this document, because it would be too long. There are manual tests which can be done by the developer or the user himself by just using the software and later reporting bugs he found and other problems. But in most cases this happens too late, because the software is usually already delivered and deployed. This makes fixing bugs and changing the problematic code a lot more difficult.

It is easier to observe the stability and reliability of a software earlier during the development process by using automated tests such as unit-, regression- and integration-tests. One development process making good use of these automated tests is **Test Driven Development** or **TDD**. TDD will be further explained in the next chapter.

2 Test Driven Development

Test Driven Development (TDD) is a software development and design paradigm which centers very specific tests to be the indicator for the development process [see Bec04]. The test-cases which represent the requirements for the tested class are specified and integrated before the actual code that needs to be tested is implemented. Necessary specifications such as class- and method names are made at the beginning of the development cycle for a specific class. Every single test should fail at first, because the tested methods won't do the work they need to, because they aren't implemented yet. After that the actual code is implemented and will be further corrected and developed until it satisfies the requirements and all tests turn green, when they are run again. [see WMV03].

In TDD it is not allowed or recommended to integrate new functionality for software when the corresponding tests do not exist yet.

The goal of this approach is improving code quality and ensuring that the code fulfills the needed requirements while slowly teaching the developers to write better testable code.

Unit Tests can be developed for almost all existing programming languages. There are a lot of different frameworks such as NUnit¹ for .NET developers or JUnit² for Java-users. In this assignment, JUnit will be used, because the code is written in Java.

¹<http://www.nunit.org/>

²<http://www.junit.org/>

3 The Point() class

This workshop contains four exercises. The first exercise required the implementation of a class `Point()` which fulfills certain requirements

The exercise was done in teams of two participants. One of them developed the class, while the other team member implemented the unit-tests. The difficult thing here was that the person who wrote the test had no clue, how his or her partner would implement the tested `Point`-class. All he or she knew was the class name and the method names. Vice versa the person who implemented the class had no idea, what the other person would test for.

This document contains the implementation of the `Point`-Class while my partner's document contains the tests. The full class can be found in the appendix of this document. Each of the following sub-chapters contain on requirement and the solution for this requirement.

3.1 The default constructor

The default constructor of the `Point` class should initialize the `x` and `y`- fields with the double values 0.0 and 0.0. The following code does just that:

```
1  public Point() {  
    this.x = 0.0;  
3   this.y = 0.0;  
   }
```

3.2 An alternate constructor

An alternate constructor should take two double values as arguments and initialize the `x` and `y`- fields with them. The following code shows how this was implemented:

```
    public Point(double x, double y) {  
2       assert(!Double.isNaN(x) && !Double.isNaN(y));  
        this.x = x;  
4       this.y = y;  
    }
```

the constructor uses an `assert()` to ensure that the field can't be initialized with arguments that are not a number. If one of the given arguments is not a valid double-value or null, the constructor will throw an `Assertion-Error`.

3.3 getters and setters

The class `Point` should provide setter and getter methods to allow other object to get access to the coordinate-fields `x` and `y`. The following code shows their implementation:

```
1  public double getX() {  
    return x;  
3  }  
  
5  public void setX(double x) {  
    assert(!Double.isNaN(x));  
7    this.x = x;  
    }  
9  
11 public double getY() {  
    return y;  
11 }  
13  
15 public void setY(double y) {  
    assert(!Double.isNaN(y));  
15    this.y = y;  
17 }
```

Once again the `assert()`-call is used in the setter-methods to ensure that the coordinate-fields can't be set to invalid values.

3.4 The `equals()` method

The `equals()` method should be overwritten so it returns `true` if the coordinates of two point objects are exactly the same. The following code shows the implementation of the `equals()` method for the class `Point`:

```
1  @Override  
    public boolean equals(Object o) {  
3      // Easiest case  
      if (o == this) {
```



```
5         return true;
6     }
7     // if the given object isn't a point, there is no need for further
8     // operations
9     if (o instanceof Point) {
10         Point p = (Point) o;
11         // Make use of the hashCode method
12         if (p.hashCode() == this.hashCode()) {
13             return true;
14         }
15         return (p.x == this.x && p.y == this.y);
16     }
17     return false;
18 }
```

This method is a little more complex than the previous ones. First it checks if the object given via the argument 'object' is exactly the same instance as the Point object it is given to. If this is true, it is the easiest case. If not, the method checks if the given object is an instance of the Point class. As this is an overwritten method inherited by the **Object**-class any instance of every other class that also inherits from **Object** can be passed via the argument. This needs to be checked. If the argument is an instance of the Point class, a cast will be performed. After that the method will use the hashCode() method (see next sub-chapter) to check if the objects are equal. At last, when the hashCode() method somehow did not do it's work properly, the equals() method will compare the coordinates of the two Point-instances.

3.5 The hashCode() method

The class Point should overwrite the hashCode method from Object. The HashCode should represent the properties of an instance of the Point class so two or more instances can be easily compared by using their hashCode() method they provide. The following code shows the over-written hashCode() method:

```
1     @Override
2     public int hashCode() {
3         // Hash should represent the point's coordinates
4         // So two points with the same coordinates should generate the same
5         // hash-value
6         return Objects.hash(this.x, this.y);
7     }
```

As the comment in the code already states, the coordinates x and y are used to generate the hash-value for an instance of the Point class.

3.6 The `toString()` method

The output of the overwritten `toString()` method should have the format ($\pm 0.0000E \pm 00, \pm 0.0000E \pm 00$). This was done by using this code:

```
1  @Override
   public String toString() {
3      NumberFormat formatter = new DecimalFormat("#.####E00");
      StringBuilder stringBuilder = new StringBuilder();
5      stringBuilder.append("(" );
      stringBuilder.append(formatter.format(this.x));
7      stringBuilder.append(", ");
      stringBuilder.append(formatter.format(this.y));
9      stringBuilder.append(" )");
      return stringBuilder.toString();
11 }
```

The class `DecimalFormat` is used here to give the output the desired format. A `StringBuilder` is used to generate the output.

3.7 The `norm()` method

The method `norm()` should return a double values which represents the distance of the instance of the `Point` class to the origin at $x=0.0$ and $y=0.0$. The following code shows how this was done:

```
1  public double norm() {
   // Pythagoras
3      return Math.sqrt(this.x * this.x + this.y * this.y);
   }
```

The distance between the point and the origin can easily be calculated using the Pythagoras-Theorem. This was done in this method.

3.8 The `rotate()` method

The method `rotate()` should rotate the point counterclockwise by an angle `theta`, given in degrees. If `theta` is bigger than 180.0° or smaller than -180.0° the method should throw an Exception named `AngleOutOfRangeException`. The following Code shows the method `rotate()`:

```
public void rotate(double theta) throws AngleOutOfRangeException {  
    // theta must be a number  
    if (Double.isNaN(theta) || theta < -180.0 || theta > 180.0) {  
        throw new AngleOutOfRangeException();  
    }  
    double[] pt = {this.x, this.y};  
    AffineTransform.getRotateInstance(Math.toRadians(theta), 0, 0)  
        .transform(pt, 0, pt, 0, 1);  
    double newX = pt[0];  
    double newY = pt[1];  
    this.x = newX;  
    this.y = newY;  
}
```

The method checks first if the given angle is a double value. If this isn't the case or the angle is not within the allowed range, a new `AngleOutOfRangeException` will be thrown.

The new coordinates can be determined by using the `AffineTransform` class from the Java AWT Framework.

The following code shows the Exception `AngleOutOfRangeException`:

```
public class AngleOutOfRangeException extends Exception {  
    private static final long serialVersionUID = 1L;  
}
```

3.9 The `displace()` method

The `displace()` method should receive another point with initialized coordinates and displace the current point by the amount of x and y of that other point. The following code does this:

```
public void displace(Point p) {  
    if(p == null){  
        return;  
    }  
    this.x += p.x;  
    this.y += p.y;  
}
```

It also covers the possibility that the given `Point p` can be null.

Figure 3.1 shows the class diagram for the `Point()` class.

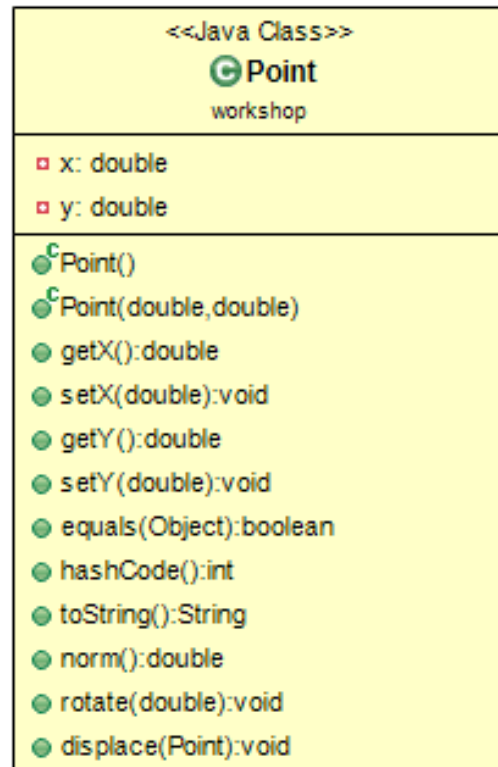
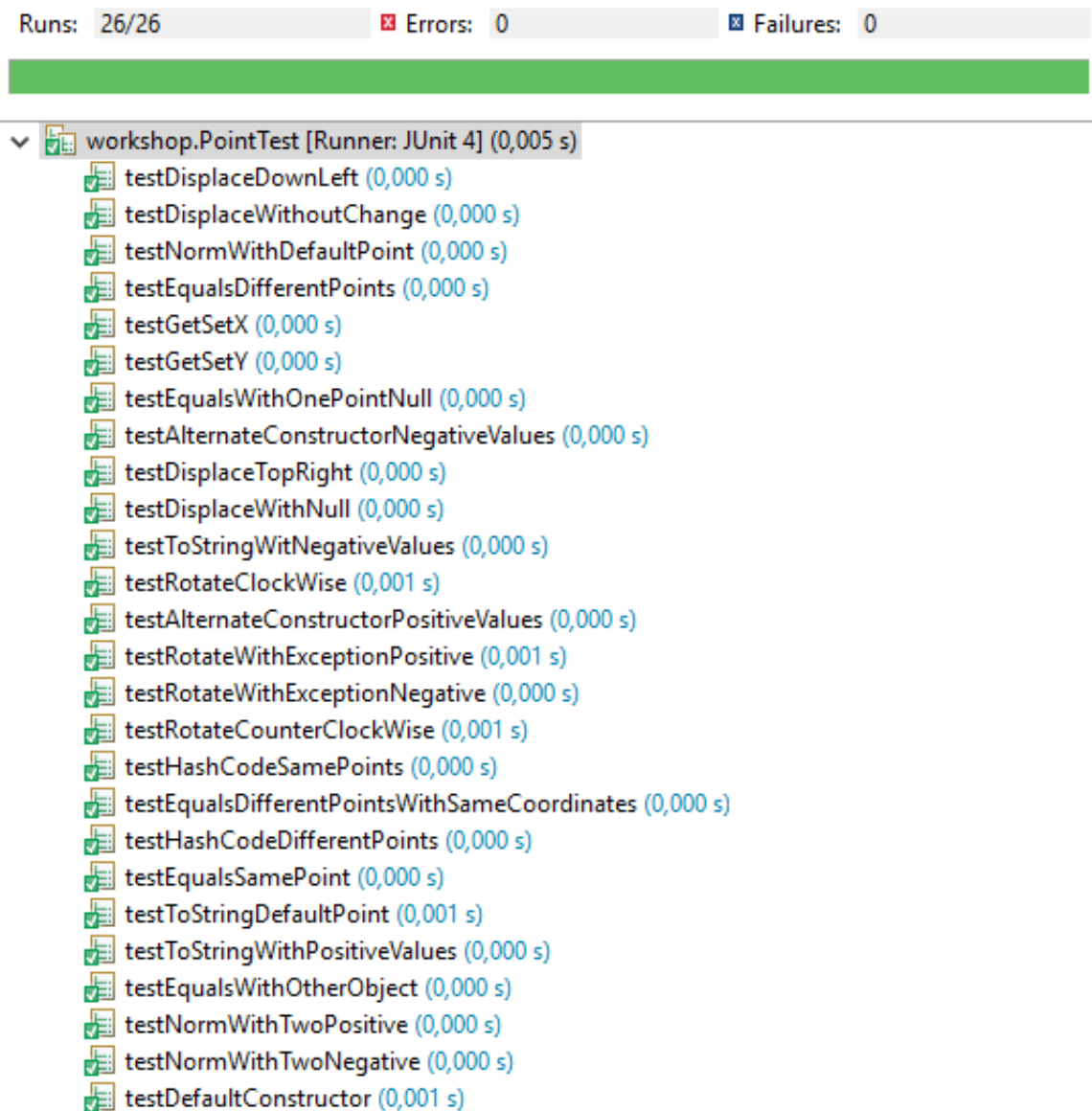
Figure 3.1: The `(Point())` class

Figure 3.2 shows the test results for the `Point()` class tested by the unit tests implemented by my lab-partner.

Figure 3.2: Test results (`Point()`)

4 Test for the *Line()* class

For the second exercise the roles were switched. The team member who wrote the tests for the *Point()* class in the first exercise implemented the new class *Line()* and the team member who implemented the *Point()* class had to write the tests for the *Line()* class.

All Tests are listed in the appendix A.0.2. The following subsection will only show the tests for the given requirements and their corresponding methods.

4.1 Constructor tests

The standard constructor for the *Line()* class should initialize a new *Line()* object without any points while an alternate constructor should receive an array of *Point*-objects to initialize the *Line* object's own *Point* array. The following tests evaluate these requirements:

```
1      @Test
2      public void testDefaultConstructor() {
3          Line line = new Line();
4          assertTrue(line.length() == 0);
5      }
6
7      @Test
8      public void testAlternateConstructor() {
9          Point[] points = getPointsExampleData();
10         Line line = new Line(points);
11         assertEquals(3, line.length());
12     }
13
14     @Test
15     public void testConstructorWithNull() {
16         Line line = new Line(null);
17         assertTrue(line.length() == 0);
18     }
```

These tests cover the functionality of the constructors. The method *getPointsExampleData()* generates some *Point*-objects for the alternate constructor's test and will be shown in the appendix A.0.2.

The *length*-method is used here to evaluate the constructor's functionality. This might be a little controversial, because an unit test should only cover one method's functionality, but

as long as the *Line* classes *Point* array is private or should at least it should be, there is no other way for observing a change in the array. Usually this isn't a problem, because the *length*-method has it's own test. As soon as there is a problem with the *length*-method the programmer will notice it.

4.2 Tests for *length()* and *add()*

The *length()* method should return an integer value which represents the current size of the array inside the *Line* object receiving the method call. The *add()* method should make it possible to add a new *Point* object to the array. The following tests should cover their functionality:

```
@Test
2 public void testAdd() {
    Line line = new Line();
4    line.add(new Point());
    assertEquals(1, line.length());
6 }

@Test
8 public void testAddWithNull() {
    Line line = new Line();
10    line.add(null);
12    assertEquals(0, line.length());
    }
14

16 public void testLength() {
    Point[] points = getPointsExampleData();
    Line line = new Line(points);
18    assertEquals(3, line.length());
    }
20

22 public void testLengthAfterAdd() {
    Point[] points = getPointsExampleData();
    Line line = new Line(points);
24    assertEquals(3, line.length());
    line.add(new Point(2.3543, 8.345));
26    assertEquals(4, line.length());
    }
```

Once again the *length()* method is used to verify the *add()* method's correct functionality.

4.3 Tests for equals()

The `equals()` method should compare two `Line` objects and determine if they share the same points. However the sequence of the points in the arrays inside the `Line` objects does not matter. The following tests evaluate this:

```
1  @Test
2  public void testEqualsWithExactlySameLineObject() {
3      Point[] points = getPointsExampleData();
4      Line line = new Line(points);
5      assertTrue(line.equals(line));
6  }
7
8  @Test
9  public void testEqualsWithTwoLineObjectsWithSamePoints() {
10     Point[] points = getPointsExampleData();
11     Line line = new Line(points);
12     Line line2 = new Line(points);
13     assertTrue(line.equals(line2));
14 }
15
16 @Test
17 public void testEqualsWithTwoLineObjectsWithSamePointsInDiffentOrder() {
18     Point[] points = getPointsExampleData();
19     Line line = new Line(points);
20     Point[] points2 = new Point[3];
21     points2[0] = new Point(2.4, -2.4);
22     points2[2] = new Point(-0.2, 8.9);
23     points2[1] = new Point(-4.4, -2.4);
24     Line line2 = new Line(points2);
25     assertTrue(line.equals(line2));
26 }
27
28 @Test
29 public void testEqualsWithTwoLinesWithDifferentPoints() {
30     Point[] points = getPointsExampleData();
31     Line line = new Line(points);
32     Point[] points2 = new Point[3];
33     points2[0] = new Point(2.3, -1.4);
34     points2[2] = new Point(-0.2, 2.9);
35     points2[1] = new Point(-42.4, -2.4);
36     Line line2 = new Line(points2);
37     assertFalse(line.equals(line2));
38 }
39
40 @Test
41 public void testEqualsWithFalseObject() {
42     Point[] points = getPointsExampleData();
```



```
43     Line line = new Line(points);
44     // comparing a Line object to a Point object should return false...
45     assertFalse(line.equals(new Point(3.9, 4.2)));
46 }
47
48 @Test
49 public void testEqualsWithNull() {
50     Point[] points = getPointsExampleData();
51     Line line = new Line(points);
52     assertFalse(line.equals(null));
53 }
```

These tests cover the requirements while also ensuring the equals-method's stability. The method should be able to handle false objects as well as null-values.

4.4 Tests for hashCode()

The hashCode()-method should generate the same hash-value for two Line-objects with the same Point array. The following tests verify the method's results:

```
1     @Test
2     public void testHashCodeWithoutPoints() {
3         Line line = new Line();
4         Line line2 = new Line();
5         assertEquals(line.hashCode(), line2.hashCode());
6     }
7
8     @Test
9     public void testHashCodeWithSamePoints() {
10        Line line = new Line(getPointsExampleData());
11        Line line2 = new Line(getPointsExampleData());
12        assertEquals(line.hashCode(), line2.hashCode());
13    }
14
15    @Test
16    public void testHashCodeWithDifferentPoints() {
17        Point[] points = getPointsExampleData();
18        Line line = new Line(points);
19        Line line2 = new Line();
20        assertNotEquals(line.hashCode(), line2.hashCode());
21    }
```

These tests should cover the hashCode()-method's functionality.

4.5 Tests for toString()

The *Line* classes' `toString()`-method should be able to print the contained points in the console in the same way, the *Line* class does. the output should look like this:

```
(( ±0.0000E ± 00, ±0.0000E ± 00 ),  
 ( ±0.0000E ± 00, ±0.0000E ± 00 ),  
 ...  
 ( ±0.0000E ± 00, ±0.0000E ± 00 )).
```

The listing below shows the tests covering the method's behavior:

```
1  @Test  
   public void testToString() {  
3      Point[] points = new Point[3];  
      points[0] = new Point(0.00000021, 12345.1246);  
5      points[1] = new Point(-0.2556785, 1.1246365);  
      points[2] = new Point(-235.25853467, -1.1246);  
7      Line line = new Line(points);  
      String expectedOutput = "(( 2.4, -2.4 ),\n ( -0.2, 8.9 ),\n ( -4.4, -2.4 ))";  
9      assertEquals(expectedOutput, line.toString());  
   }
```

There are not many problem that can occur here. So there are not too many tests for this method.

4.6 Tests for isValid()

The `isValid()` should be able to check if the slope of the regression line and the intercept with the Y-axis can be calculated. There are a few cases when this should not be possible. When the number of points in the line is either 0 or 1, the slope cannot be determined. The same happens when all points have the same coordinates or have the same x-coordinate. So here are some test which verify that the `isValid()` method can handle these cases correctly:

```
   @Test  
2   public void testIsValidWithZeroPoints() {  
       Line line = new Line();  
4       assertFalse(line.isValid());  
   }  
  
6   @Test  
8   public void testIsValidWithOnePoint() {
```

```
10     Line line = new Line();
11     line.add(new Point(2.1, 4.3));
12     assertFalse(line.isValid());
13 }
14
15 @Test
16 public void testIsValidWithManyPoints() throws RegressionFailedException {
17     Point[] points = getPointsExampleData();
18     Line line = new Line(points);
19     assertTrue(line.isValid());
20 }
21
22 @Test
23 public void testIsValidWithTwoPointsWithSameCoordinates() {
24     Point[] points = new Point[2];
25     points[0] = new Point(2.1, 2.1);
26     points[1] = new Point(2.1, 2.1);
27     Line line = new Line(points);
28     assertFalse(line.isValid());
29 }
30
31 @Test
32 public void testIsValidWithTwoPointsWithSameXCoordinates() {
33     Point[] points = new Point[2];
34     points[0] = new Point(2.1, 23.34);
35     points[1] = new Point(2.1, 21.23);
36     Line line = new Line(points);
37     assertFalse(line.isValid());
38 }
```

These tests do not evaluate the output for slope and intercept, as there are different methods for that.

4.7 Tests for slope()

The `slope()` method should be able to calculate the linear regression line for the points contained inside the `Line` object. If the slope cannot be calculated a new `RegressionFailedException` must be thrown. The following tests check the functionality for this method:

```
1     @Test
2     public void testSlopeWithTwoValidPoints() throws RegressionFailedException{
3         // very simple tests
4         Point[] points = new Point[2];
5         points[0] = new Point(0, 0);
6         points[1] = new Point(1, 1);
```

```
7      Line line = new Line(points);
      double slope = line.slope();
9      assertEquals(1, slope, 0);
  }

11

  @Test
13  public void testSlopeWithValidPoints() throws RegressionFailedException{
      // more complex parameters
15      Point[] points = new Point[6];
      points[0] = new Point(0.2, 0);
17      points[1] = new Point(1, 3.12);
      points[2] = new Point(3.23, 1);
19      points[3] = new Point(-1.5, 1.3);
      points[4] = new Point(4.9, 12.2);
21      points[5] = new Point(3.32, 2);
      Line line = new Line(points);
23      double slope = line.slope();
      assertEquals(1.2, slope, 0.03);
25  }

27  @Test(expected = RegressionFailedException.class)
  public void testSlopeWithSamePoints() throws RegressionFailedException{
29      Point[] points = new Point[2];
      points[0] = new Point(0, 0);
31      points[1] = new Point(0, 0);
      // these points have the same coordinates.
33      // So the slope can't be calculated with these two.
      Line line = new Line(points);
35      line.slope();
  }
```

The first test works with very simple points evaluating the very functionality of the `slope` method by using Points that generate a regression line with the value '1' as slope. The second test uses some more complex point and check the result. The third test uses one of the cases discussed in the `isValid()` section to force the occurrence of a `RegressionFailedException`.

4.8 Tests for `intercept()`

the method `intercept()` should determine the interception of the regression line with the Y-axis. There are some cases, where this cannot be done. Then a new `RegressionFailedException` should be thrown. When the all the points in the line have the same x-coordinate, the slope is infinite and there can never be an interception with the y-axis. The following tests evaluate the behavior of the method `intercept()`:

```

@Test
2  public void testInterceptIWithValidPoints() throws RegressionFailedException{
    Point[] points = new Point[2];
4   points[0] = new Point(0, 0);
    points[1] = new Point(1, 1);
6   Line line = new Line(points);
    double intercept = line.intercept();
8   assertEquals(0, intercept, 0);
}

@Test(expected = RegressionFailedException.class)
12 public void testInterceptIWithInvalidPoints() throws RegressionFailedException{
    Point[] points = new Point[2];
14 points[0] = new Point(0, 0);
    points[1] = new Point(0, 1);
16 // with these points the line lies on the y-axis. therefore slope can't be calculated.
    // And without slope, there is no intercept.
18 Line line = new Line(points);
    line.intercept();
20 }

```

Again there is one test for the normal behavior of the `intercept()` method and another test for one of the cases discussed above where intercept cannot be determined and a `RegressionFailedException` is expected.

In the following (Figure 4.1) the results of the tests created for the line class are shown.

Finally figure 4.2 shows the UML diagram of the `Line()` class.

Finished after 0,408 seconds

Runs: 26/26

Errors: 0

Failures: 0

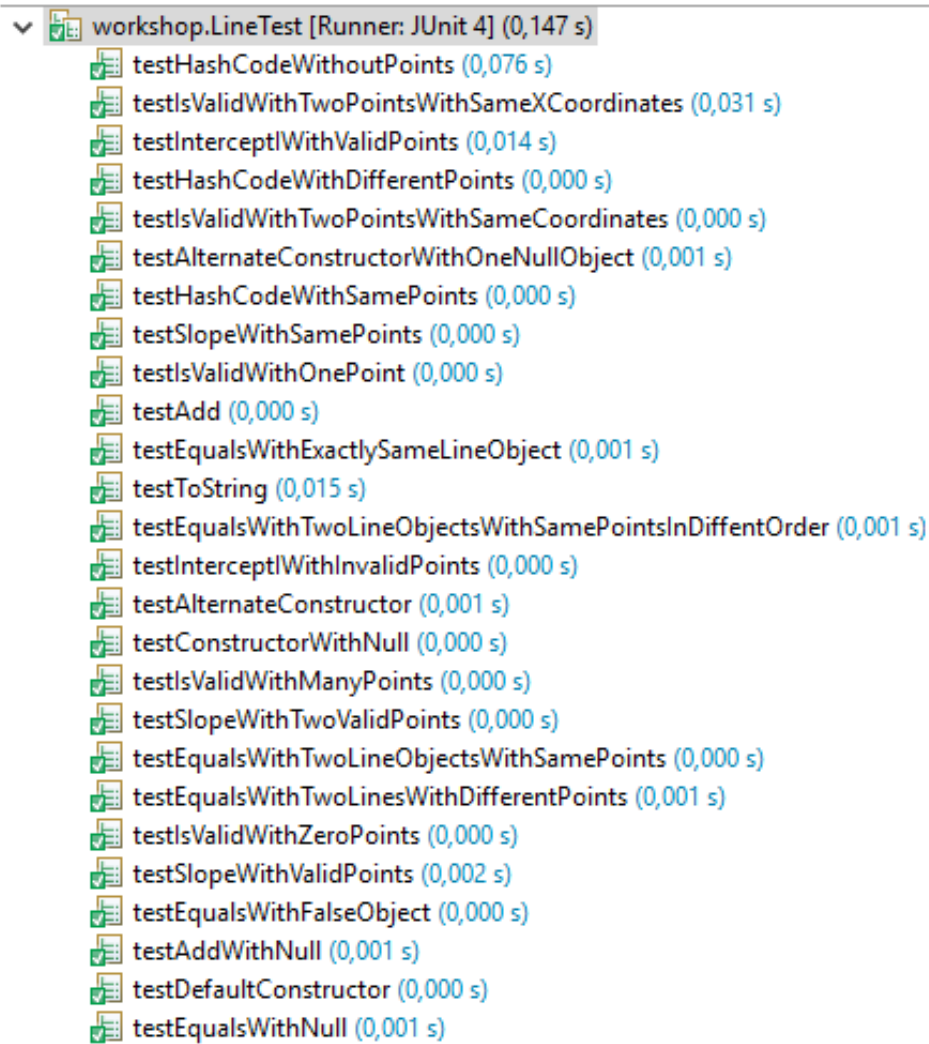
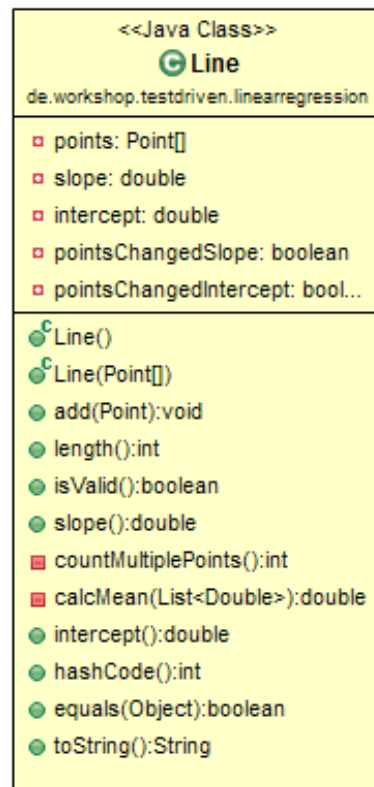
- 
- The screenshot displays the JUnit 4 test results for the `workshop.LineTest` class. The tests are organized in a tree view under the heading `workshop.LineTest [Runner: JUnit 4] (0,147 s)`. Each test is represented by a green checkmark icon, indicating a successful pass. The tests and their durations are as follows:
- `testHashCodeWithoutPoints (0,076 s)`
 - `testIsValidWithTwoPointsWithSameXCoordinates (0,031 s)`
 - `testInterceptWithValidPoints (0,014 s)`
 - `testHashCodeWithDifferentPoints (0,000 s)`
 - `testIsValidWithTwoPointsWithSameCoordinates (0,000 s)`
 - `testAlternateConstructorWithOneNullObject (0,001 s)`
 - `testHashCodeWithSamePoints (0,000 s)`
 - `testSlopeWithSamePoints (0,000 s)`
 - `testIsValidWithOnePoint (0,000 s)`
 - `testAdd (0,000 s)`
 - `testEqualsWithExactlySameLineObject (0,001 s)`
 - `testToString (0,015 s)`
 - `testEqualsWithTwoLineObjectsWithSamePointsInDiffentOrder (0,001 s)`
 - `testInterceptWithInvalidPoints (0,000 s)`
 - `testAlternateConstructor (0,001 s)`
 - `testConstructorWithNull (0,000 s)`
 - `testIsValidWithManyPoints (0,000 s)`
 - `testSlopeWithTwoValidPoints (0,000 s)`
 - `testEqualsWithTwoLineObjectsWithSamePoints (0,000 s)`
 - `testEqualsWithTwoLinesWithDifferentPoints (0,001 s)`
 - `testIsValidWithZeroPoints (0,000 s)`
 - `testSlopeWithValidPoints (0,002 s)`
 - `testEqualsWithFalseObject (0,000 s)`
 - `testAddWithNull (0,001 s)`
 - `testDefaultConstructor (0,000 s)`
 - `testEqualsWithNull (0,001 s)`

Figure 4.1: Test results (*Line()*)

Figure 4.2: UML diagram of the `Line()` class

5 The LinearRegression Application

The third exercise should be solved individually by each student. There were two files with example data provided (data_short.dat and data_long.dat) as well as a java library 'readFile.jar' containing classes that could read the data inside these two files. An example code on how to use the library was provided too.

The two .dat files contained multiple sequences of double-pairs which could be parsed into `Point` and `Line` Objects. This was the subject of this exercise. Both files should be parsed by the program and the results should be analyzed to get the following statistics for the given data:

- The total number of lines, valid and invalid counted separately.
- The average number of points per valid lines.
- The average slope and its standard deviation for all valid lines.
- The average y-intercept and its standard deviation for all valid lines.

Figure 5.1 shows the results for the lines from file data_short.dat and figure 5.2 the results

```
run time is 1128 milliseconds

Statistics for file: data_short.dat
Total number of valid lines: 10
Total number of invalid lines: 0
Average number of points per valid line: 12.8
Average Slope for Valid Lines: 0.7478633907788373
with standard deviation: 0.005462144108110208
Average intercept for valid lines: -1.916502262038548
with standard deviation: 0.007847149591650194
```

Figure 5.1: Results from data_short.dat

from data_long.dat.


```
run time is 433642 milliseconds

Statistics for file: data_long.dat
Total number of valid lines: 9618
Total number of invalid lines: 392
Average number of points per valid line: 13.552297775005199
Average Slope for Valid Lines: 0.7492665051561573
with standard deviation: 0.00473897619938974
Average intercept for valid lines: -1.9180493271694963
with standard deviation: 0.008197579816654631
```

Figure 5.2: Results from data_long.dat

The code for the `LinearRegression` Application will be shown in the appendix A.0.3. The classes' name is `LinearRegressionProgram`. Figure 5.3 shows the UML diagram for the complete program and figure 5.4 shows the sequence diagram for the application.

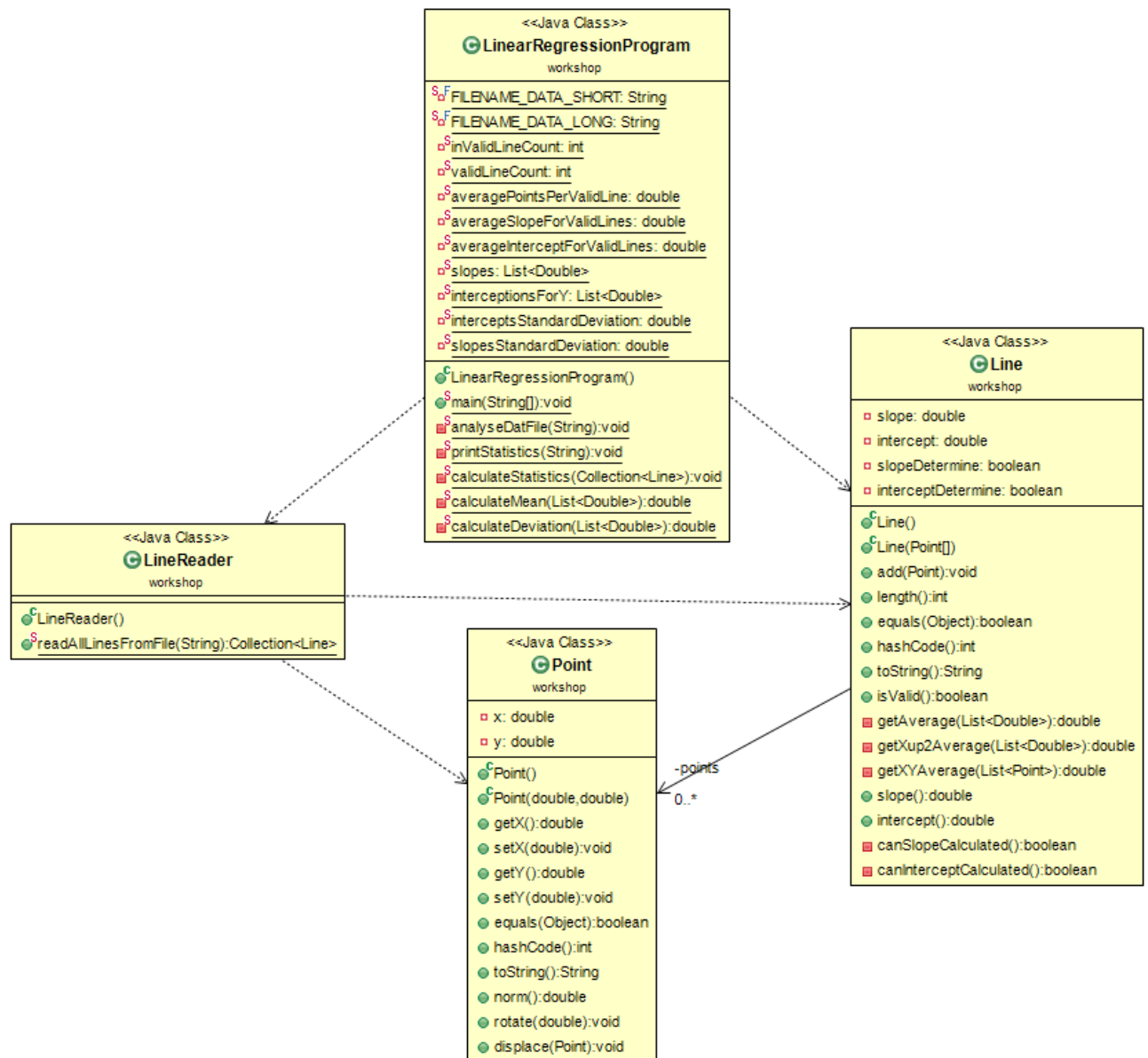


Figure 5.3: UML class diagram for the LinearRegression application

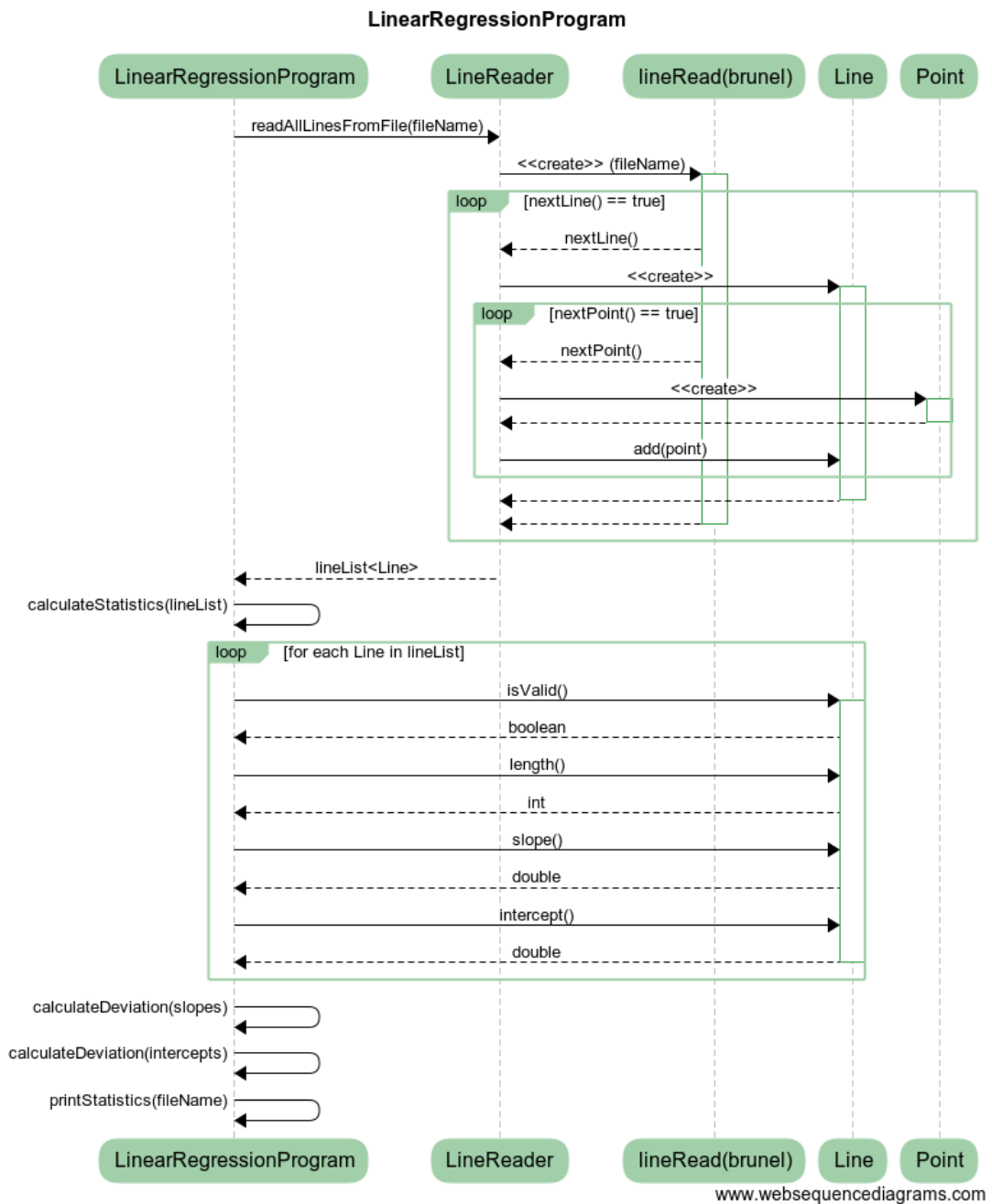


Figure 5.4: Sequence diagram for the LinearRegression application

6 Evaluating the performance

In last exercise of this workshop the time the program needed to read in the data and the time to perform the required operations with the number of points in the data sets should be evaluated.

To do this, the provided example code for time calculations were used to get the following data for the file `data_short.dat`:

```
Times stopped for file: C:\Users\User\Desktop\TDD\TDDAssignment\TDD\src\main\java\workshop\data_long.dat
programm runtime in milliseconds: 446100
reading dat-files runtime in milliseconds: 442985
analyzing data runtime in milliseconds: 2191
printing and parsing results runtime in milliseconds: 924
```

Figure 6.1: Stopped durations for the different operation in the `LinearRegression` application in milliseconds

This shows that the time the program needs to read the data from `data_short.dat` is a lot longer than the duration stopped for the analyzing process. Therefore the reading time and how it depends on the number of points in the lines were further investigated by using the following `HashMap` in the `LineReader` class:

```
private static Map<Integer, List<Long>> timeData = new HashMap<Integer, List<Long>>();
```

At the start of each run through the loop reading a line from the `dat` file, a timestamp was initialized. The same was done at the end of each run through the loop. After that the duration between those timestamps was calculated and saved into the `HashMap` using this code:

```
1 // Get the duration it took, to read a single line in milliseconds.
2 long timeToReadLine = endLineTimeStamp.getTime() - startLineTimeStamp.getTime();
3 if (timeData.containsKey(line.length())) {
4     timeData.get(line.length()).add(timeToReadLine);
5 } else {
6     List<Long> timeList = new ArrayList<>();
7     timeList.add(timeToReadLine);
8     timeData.put(line.length(), timeList);
9 }
}
```

The number of points is used as key while the duration is added to the associated list of durations. Using this resulted in a lot of time data for each line length, which had to be normalized

in the main application using the following method:

```
2 // Gets the avarage for the time it took to process a line with a certain
// amount of points from raw timestamp-data
private static Map<Integer, Long> normalizeTimeData(Map<Integer, List<Long>> data){
4     Map<Integer, Long> normalizedData = new HashMap<Integer, Long>();
    for (Integer key : data.keySet()){
6         List<Long> stoppedTimes = data.get(key);
        long avarage = 0;
8         long sum = 0;
        for (Long time : stoppedTimes){
10             sum += time;
        }
12         if(stoppedTimes.size() != 0){
            avarage = (long) sum / stoppedTimes.size();
14         }
        normalizedData.put(key, avarage);
16     }
    return normalizedData;
18 }
```

This method's result is parsed into an Excel file in the project path. the data from the Excel file is shown in the table 6.1. The class responsible for writing the Excel File is listed in the appendix A.0.5.

Additionally, Figure 6.2 shows the plot of time needed to read and create a line over the number of point in this line from the Excel file. It can be seen that the time increases fairly linear with number of point. But there also is a leap from nine points to ten. This must be caused by the class `lineRead` which was provided to reading in the data.

Number of Points	Time to read (ms)
1	2
2	5
3	7
4	10
5	12
6	15
7	18
8	20
9	23
10	34
11	38
12	42
13	45
14	49
15	52
16	56
17	60
18	63
19	66
20	70
21	73
22	77
23	80
24	84
25	87

Table 6.1: Number of points and their times

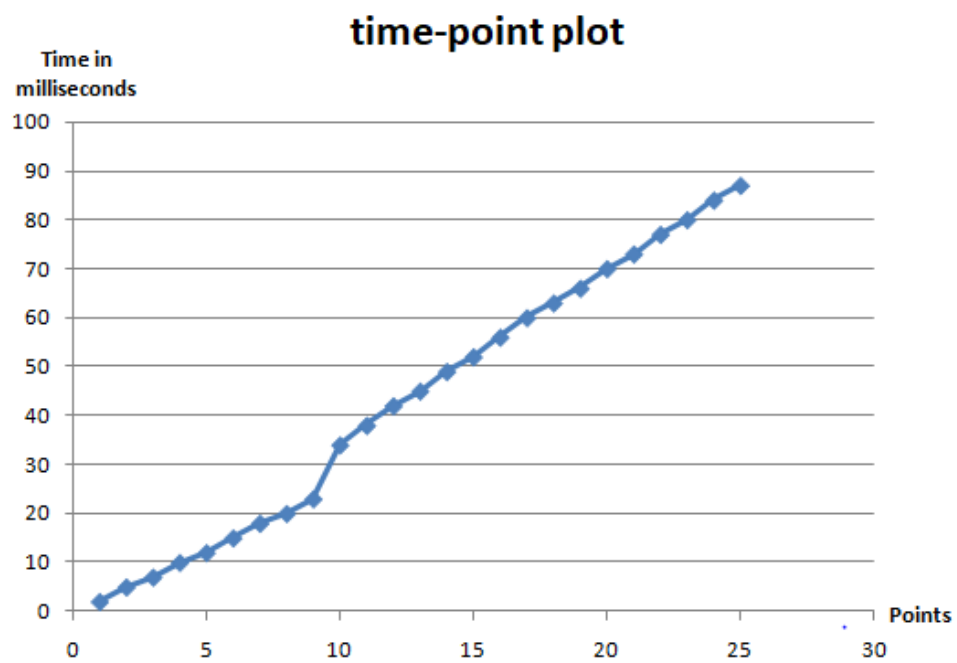


Figure 6.2: Time-Point plot

Bibliography

- [Bec04] Kent Beck. *Test-Driven development by Example*. Pearson, 2004. ISBN: 8131715957 (cit. on p. 2).
- [WMV03] Laurie Williams, E. Michael Maximilien, and Mladen Vouk. “Test-Driven Development as a Defect-Reduction Practice”. In: *14th IEEE International Symposium on Software Reliability Engineering ISSRE 2003*. Denver, Colorado: IEEE, 2003. URL: <http://ieeexplore.ieee.org/document/1251029/> (visited on 10/01/2016) (cit. on p. 2).

A Appendix

A.0.1 Point class listing

```
package workshop;

import java.awt.geom.AffineTransform;
import java.util.Locale;
import java.util.Objects;

public class Point {
    private double x;
    private double y;

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

    public Point(double x, double y) {
        assert (!Double.isNaN(x) && !Double.isNaN(y));
        this.x = x;
        this.y = y;
    }

    public double getX() {
        return x;
    }

    public void setX(double x) {
        assert (!Double.isNaN(x));
        this.x = x;
    }

    public double getY() {
        return y;
    }

    public void setY(double y) {
        assert (!Double.isNaN(y));
        this.y = y;
    }

    @Override
    public boolean equals(Object o) {
        // Easiest case
        if (o == this) {
            return true;
        }
        // if the given object isn't a point, there is no need for further
        // operations
        if (o instanceof Point) {
```

```

        Point p = (Point) o;
        // Make use of the hashCode method
        if (p.hashCode() == this.hashCode()) {
            return true;
        }
        return (p.x == this.x && p.y == this.y);
    }
    return false;
}

@Override
public int hashCode() {
    // Hash should represent the point's coordinates
    // So two points with the same coordinates should generate the same
    // hash-value
    return Objects.hash(this.x, this.y);
}

@Override
public String toString() {
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.append("(" );
    stringBuilder.append(String.format(Locale.ROOT, "%+2.4E", this.x));
    stringBuilder.append(", ");
    stringBuilder.append(String.format(Locale.ROOT, "%+2.4E", this.y));
    stringBuilder.append(")");
    return stringBuilder.toString();
}

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

public void rotate(double theta) throws AngleOutOfRangeException {
    // theta must be an number
    if (Double.isNaN(theta) || theta < -180.0 || theta > 180.0) {
        throw new AngleOutOfRangeException();
    }
    double[] pt = {this.x, this.y};
    AffineTransform.getRotateInstance(Math.toRadians(theta), 0, 0)
        .transform(pt, 0, pt, 0, 1);
    double newX = pt[0];
    double newY = pt[1];
    this.x = newX;
    this.y = newY;
}

public void displace(Point p) {
    if(p == null){

```

```
100         return;
101     }
102     this.x += p.x;
103     this.y += p.y;
104 }
105 }
```

A.0.2 LineTest class listing

```
package workshop;

import static org.junit.Assert.*;

import org.junit.Test;

public class LineTest {

    //-----
    // generates some points for different tests
    //-----

    private Point[] getPointsExampleData() {
        Point[] points = new Point[3];
        points[0] = new Point(2.4, -2.4);
        points[1] = new Point(-0.2, 8.9);
        points[2] = new Point(-4.4, -2.4);
        return points;
    }

    //-----
    // Constructor Tests
    //-----

    @Test
    public void testDefaultConstructor() {
        Line line = new Line();
        assertTrue(line.length() == 0);
    }

    @Test
    public void testAlternateConstructor() {
        Point[] points = getPointsExampleData();
        Line line = new Line(points);
        assertEquals(3, line.length());
    }

    @Test
    public void testAlternateConstructorWithOneNullObject() {
        Point[] points = getPointsExampleData();
        points[0] = null;
        Line line = new Line(points);
        assertEquals(2, line.length());
    }

    @Test
    public void testConstructorWithNull() {
        Line line = new Line(null);
    }
}
```

```
        assertTrue(line.length() == 0);
50    }

52    //-----
54    // length() and add() Tests
56    //-----

    @Test
    public void testAdd() {
58        Line line = new Line();
        line.add(new Point());
60        assertEquals(1, line.length());
    }

62

    @Test
64    public void testAddWithNull() {
        Line line = new Line();
66        line.add(null);
        assertEquals(0, line.length());
68    }

70    public void testLength() {
        Point[] points = getPointsExampleData();
72        Line line = new Line(points);
        assertEquals(3, line.length());
74    }

76    public void testLengthAfterAdd() {
        Point[] points = getPointsExampleData();
78        Line line = new Line(points);
        assertEquals(3, line.length());
80        line.add(new Point(2.3543, 8.345));
        assertEquals(4, line.length());
82    }

84    //-----
86    // equals() Tests
88    //-----

    @Test
    public void testEqualsWithExactlySameLineObject() {
90        Point[] points = getPointsExampleData();
        Line line = new Line(points);
92        assertTrue(line.equals(line));
    }

94

    @Test
96    public void testEqualsWithTwoLineObjectsWithSamePoints() {
        Point[] points = getPointsExampleData();
98        Line line = new Line(points);
```

```

    Line line2 = new Line(points);
    assertTrue(line.equals(line2));
}

@Test
public void testEqualsWithTwoLineObjectsWithSamePointsInDiffentOrder() {
    Point[] points = getPointsExampleData();
    Line line = new Line(points);
    Point[] points2 = new Point[3];
    points2[0] = new Point(2.4, -2.4);
    points2[2] = new Point(-0.2, 8.9);
    points2[1] = new Point(-4.4, -2.4);
    Line line2 = new Line(points2);
    assertTrue(line.equals(line2));
}

@Test
public void testEqualsWithTwoLinesWithDifferentPoints() {
    Point[] points = getPointsExampleData();
    Line line = new Line(points);
    Point[] points2 = new Point[3];
    points2[0] = new Point(2.3, -1.4);
    points2[2] = new Point(-0.2, 2.9);
    points2[1] = new Point(-42.4, -2.4);
    Line line2 = new Line(points2);
    assertFalse(line.equals(line2));
}

@Test
public void testEqualsWithFalseObject() {
    Point[] points = getPointsExampleData();
    Line line = new Line(points);
    // comparing a Line object to a Point object should return false...
    assertFalse(line.equals(new Point(3.9, 4.2)));
}

@Test
public void testEqualsWithNull() {
    Point[] points = getPointsExampleData();
    Line line = new Line(points);
    assertFalse(line.equals(null));
}

//-----
// hashCode() Tests
//-----

@Test
public void testHashCodeWithoutPoints() {
    Line line = new Line();

```

```

    Line line2 = new Line();
150    assertEquals(line.hashCode(), line2.hashCode());
    }

152    @Test
154    public void testHashCodeWithSamePoints() {
        Line line = new Line(getPointsExampleData());
156        Line line2 = new Line(getPointsExampleData());
        assertEquals(line.hashCode(), line2.hashCode());
158    }

160    @Test
162    public void testHashCodeWithDifferentPoints() {
        Point[] points = getPointsExampleData();
        Line line = new Line(points);
164        Line line2 = new Line();
        assertEquals(line.hashCode(), line2.hashCode());
166    }

168    //-----
169    // toString() Tests
170    //-----

172    @Test
174    public void testToString() {
        Point[] points = new Point[3];
        points[0] = new Point(0.00000021, 12345.1246);
176        points[1] = new Point(-0.2556785, 1.1246365);
        points[2] = new Point(-235.25853467, -1.1246);
178        Line line = new Line(points);
        String expectedOutout = "(( +2.1000E-07, +1.2345E+04 ),\n ( -2.5568E-01, +1.1246E+00 ),\n (
            -2.3526E+02, -1.1246E+00 ))";
180        assertEquals(expectedOutout, line.toString());
    }

182    //-----
183    // isValid() Tests
184    //-----

186    @Test
188    public void testIsValidWithZeroPoints() {
        Line line = new Line();
190        assertFalse(line.isValid());
    }

192    @Test
194    public void testIsValidWithOnePoint() {
        Line line = new Line();
196        line.add(new Point(2.1, 4.3));
        assertFalse(line.isValid());
    }

```



```

198     }

200     @Test
201     public void testIsValidWithManyPoints() throws RegressionFailedException {
202         Point[] points = getPointsExampleData();
203         Line line = new Line(points);
204         assertTrue(line.isValid());
205     }

206

207     @Test
208     public void testIsValidWithTwoPointsWithSameCoordinates() {
209         Point[] points = new Point[2];
210         points[0] = new Point(2.1, 2.1);
211         points[1] = new Point(2.1, 2.1);
212         Line line = new Line(points);
213         assertFalse(line.isValid());
214     }

215

216     @Test
217     public void testIsValidWithTwoPointsWithSameXCoordinates() {
218         Point[] points = new Point[2];
219         points[0] = new Point(2.1, 23.34);
220         points[1] = new Point(2.1, 21.23);
221         Line line = new Line(points);
222         assertFalse(line.isValid());
223     }

224     //-----
225     // Slope Tests
226     //-----

227

228     @Test
229     public void testSlopeWithTwoValidPoints() throws RegressionFailedException{
230         // very simple tests
231         Point[] points = new Point[2];
232         points[0] = new Point(0, 0);
233         points[1] = new Point(1, 1);
234         Line line = new Line(points);
235         double slope = line.slope();
236         assertEquals(1, slope, 0);
237     }

238

239     @Test
240     public void testSlopeWithValidPoints() throws RegressionFailedException{
241         // more complex parameters
242         Point[] points = new Point[6];
243         points[0] = new Point(0.2, 0);
244         points[1] = new Point(1, 3.12);
245         points[2] = new Point(3.23, 1);
246         points[3] = new Point(-1.5, 1.3);
247         points[4] = new Point(4.9, 12.2);

```

```

248     points[5] = new Point(3.32, 2);
        Line line = new Line(points);
250     double slope = line.slope();
        assertEquals(1.2, slope, 0.03);
252 }

@Test(expected = RegressionFailedException.class)
    public void testSlopeWithSamePoints() throws RegressionFailedException{
256     Point[] points = new Point[2];
        points[0] = new Point(0, 0);
258     points[1] = new Point(0, 0);
        // these points have the same coordinates.
260     // So the slope can't be calculated with these two.
        Line line = new Line(points);
262     line.slope();
    }

    //-----
    // Intercept Tests
    //-----

268     @Test
    public void testInterceptIWithValidPoints() throws RegressionFailedException{
270         Point[] points = new Point[2];
        points[0] = new Point(0, 0);
272         points[1] = new Point(1, 1);
        Line line = new Line(points);
274         double intercept = line.intercept();
        assertEquals(0, intercept, 0);
276     }

278     @Test(expected = RegressionFailedException.class)
    public void testInterceptIWithInvalidPoints() throws RegressionFailedException{
280         Point[] points = new Point[2];
        points[0] = new Point(0, 0);
282         points[1] = new Point(0, 1);
        // with these points the line lies on the y-axis. therefore slope can't be calculated.
284         // And without slope, there is no intercept.
        Line line = new Line(points);
286         line.intercept();
288     }
}

```

A.0.3 LinearRegressionProgram class listing

```

1 package workshop;

3 import java.io.IOException;
  import java.util.ArrayList;
5 import java.util.Collection;
  import java.util.Date;
7 import java.util.HashMap;
  import java.util.List;
9 import java.util.Map;

11 public class LinearRegressionProgram {

13     // Files
    private static final String FILENAME_DATA_SHORT = "C:\\Users\\User\\Desktop\\TDD\\TDDAssignment\\TDD\\src\\main\\java\\workshop\\data_short.dat";
15     private static final String FILENAME_DATA_LONG = "C:\\Users\\User\\Desktop\\TDD\\TDDAssignment\\TDD\\src\\main\\java\\workshop\\data_long.dat";
    private static final String FILENAME_READER_EXCEL = "ReaderData.xlsx";
17     private static final String FILENAME_ANALYZER_EXCEL = "AnalyzerData.xlsx";

19     // Data to calculate
    private static int invalidLineCount = 0;
21     private static int validLineCount = 0;
    private static double averagePointsPerValidLine = 0;
23     private static double averageSlopeForValidLines = 0;
    private static double averageInterceptForValidLines = 0;
25     private static List<Double> slopes = new ArrayList<Double>();
    private static List<Double> interceptionsForY = new ArrayList<Double>();
27     private static double interceptsStandardDeviation = 0;
    private static double slopesStandardDeviation = 0;
29     private static Map<Integer, List<Long>> analyzerDataForCalculations;

31     // Times stopped in milliseconds
    private static long overallTime;
33     private static long readTime;
    private static long analyzeTime;
35     private static long printAndParseResultsTime;

37     public static void main(String[] args) {
        Date startOverallTimestamp = new Date();
39        analyseDatFile(FILENAME_DATA_LONG);
        Date endOverallTimestamp = new Date();
41        overallTime = endOverallTimestamp.getTime() - startOverallTimestamp.getTime();
        printTimes(FILENAME_DATA_LONG);
43    }

45     private static void analyseDatFile(final String fileName) {
        analyzerDataForCalculations = new HashMap<Integer, List<Long>>();

```

```

47     Date startReadTimestamp = new Date();
    Collection<Line> linesFromFile = LineReader.readAllLinesFromFile(fileName);
49     Date endReadTimestamp = new Date();
    readTime = endReadTimestamp.getTime() - startReadTimestamp.getTime();
51     Date startAnalyzeTimestamp = new Date();
    calculateStatistics(linesFromFile);
53     Date endAnalyzeTimestamp = new Date();
    analyzeTime = endAnalyzeTimestamp.getTime() - startAnalyzeTimestamp.getTime();
55     Date startPrintAndParseResultsTimestamp = new Date();
    printStatistics(fileName);
57     Map<Integer, Long> normalizedReaderData = normalizeTimeData(LineReader.getAnalyzerData());
    Map<Integer, Long> normalizedAnalyzerData = normalizeTimeData(analyzerDataForCalculations);
59     try {
        ExcelWriter.writeExcelOutput(normalizedReaderData, FILENAME_READER_EXCEL);
61         ExcelWriter.writeExcelOutput(normalizedAnalyzerData, FILENAME_ANALYZER_EXCEL);
    } catch (IOException e) {
63         System.err.println("Excel Creation failed !");
        e.printStackTrace();
65     }
    Date endPrintAndParseResultsTimestamp = new Date();
67     printAndParseResultsTime = endPrintAndParseResultsTimestamp.getTime() -
        startPrintAndParseResultsTimestamp.getTime();
    }

69     // print-method for statistics outputs
71     private static void printStatistics(String fileName) {
        System.out.println("\nStatistics for file: "+fileName);
73         System.out.println("Total number of valid lines: " + validLineCount);
        System.out.println("Total number of invalid lines: " + invalidLineCount);
75         System.out.println("Average number of points per valid line: " + averagePointsPerValidLine);
        ;
        System.out.println("Average Slope for Valid Lines: " + averageSlopeForValidLines);
77         System.out.println("with standard deviation: " + slopesStandardDeviation);
        System.out.println("Average intercept for valid lines: " + averageInterceptForValidLines);
79         System.out.println("with standard deviation: " + interceptsStandardDeviation);
    }

81     // print-method for time outputs
83     private static void printTimes(String fileName) {
        System.out.println("\n\nTimes stopped for file: "+fileName);
85         System.out.println("programm runtime in milliseconds: " + overallTime);
        System.out.println("reading dat-files runtime in milliseconds: " + readTime);
87         System.out.println("analyzing data runtime in milliseconds: " + analyzeTime);
        System.out.println("printing and parsing results runtime in milliseconds: " +
            printAndParseResultsTime);
89     }

91     // analyses all lines for required data
    private static void calculateStatistics(Collection<Line> linesFromFile) {
93         double slopeSum = 0.0;

```

```

double interceptSum = 0.0;
95 int pointSum = 0;

97 for (Line line : linesFromFile) {
    Date startLine = new Date();
99     if (line.isValid()) {
        validLineCount++;
101        pointSum += line.length();
        try {
103            double slope = line.slope();
            slopeSum += slope;
105            slopes.add(slope);
            double intercept = line.intercept();
107            interceptSum += intercept;
            interceptionsForY.add(intercept);
109        } catch (RegressionFailedException e) {
            e.printStackTrace();
111        }
        } else {
113            invalidLineCount++;
        }
115    Date endLine = new Date();
    // Get the amount of time it took, to calculate statistics for a line.
117    long timeToReadLine = endLine.getTime() - startLine.getTime();
    if (analyzerDataForCalculations.containsKey(line.length())) {
119        analyzerDataForCalculations.get(line.length()).add(timeToReadLine);
    } else {
121        List<Long> timeList = new ArrayList<>();
        timeList.add(timeToReadLine);
123        analyzerDataForCalculations.put(line.length(), timeList);
    }
125 }

// Dividing by 0 destroys reality
127 if (validLineCount > 0) {
    averagePointsPerValidLine = (double) pointSum / validLineCount;
129    averageSlopeForValidLines = (double) slopeSum / validLineCount;
    averageInterceptForValidLines = (double) interceptSum / validLineCount;
131 }
    slopesStandardDeviation = calculateDeviation(slopes);
133    interceptsStandardDeviation = calculateDeviation(interceptionsForY);
}

135

// Calculates the mean for a List of double-values
137 private static double calculateMean(List<Double> list) {
    return list.stream().mapToDouble(p -> p.doubleValue()).sum() / list.size();
139 }

141 // Calculates the standard deviation for a list of double-values
private static double calculateDeviation(List<Double> list) {
143     double temp = 0;

```

```
145     for (Double double1 : list) {  
146         temp += Math.pow((double1.doubleValue() - calculateMean(list)), 2);  
147     }  
148     return Math.sqrt(temp / list.size());  
149 }  
  
150 // Gets the avarage for the time it took to process a line with a certain  
151 // amount of points from raw timestamp-data  
152 private static Map<Integer, Long> normalizeTimeData(Map<Integer, List<Long>>> data){  
153     Map<Integer, Long> normalizedData = new HashMap<Integer, Long>();  
154     for (Integer key : data.keySet()){  
155         List<Long> stoppedTimes = data.get(key);  
156         long avarage = 0;  
157         long sum = 0;  
158         for (Long time : stoppedTimes){  
159             sum += time;  
160         }  
161         if(stoppedTimes.size() != 0){  
162             avarage = (long) sum / stoppedTimes.size();  
163         }  
164         normalizedData.put(key, avarage);  
165     }  
166     return normalizedData;  
167 }  
}
```

A.0.4 Line Reader listing

```

package workshop;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Date;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import uk.ac.brunel.ee.lineRead;
import uk.ac.brunel.ee.UnreadException;
import uk.ac.brunel.ee.RereadException;

public class LineReader {

    private static Map<Integer, List<Long>> timeData = new HashMap<Integer, List<Long>>();

    public static Map<Integer, List<Long>> getAnalyzerData() {
        Map<Integer, List<Long>> analyzerDataCopy = timeData;
        return analyzerDataCopy;
    }

    /**
     * @param fileName
     *         the filename
     */
    public static Collection<Line> readAllLinesFromFile(String fileName) {
        // new Analysis
        timeData = new HashMap<Integer, List<Long>>();

        List<Line> lines = new ArrayList<Line>();

        Date start = new Date();

        // Open the file and initialise
        lineRead reader = new lineRead(fileName);

        // Loop over all the lines in the data set
        while (reader.nextLine()) {
            Date startLineTimeStamp = new Date();
            Line line = new Line();
            boolean np = true;
            // Loop over all the points associated with the current line
            while (np) {
                try {
                    np = reader.nextPoint();
                } catch (UnreadException UE) {

```

```

        System.out.println(UE);
50        System.exit(0);
    }
52    // If there is another point read it.
    if (np) {
54        try {
            double x = reader.getX();
56            double y = reader.getY();
            Point point = new Point(x, y);
58            line.add(point);
        } catch (RereadException RE) {
60            System.out.println(RE);
            System.exit(0);
62        }
    }
64    }
    lines.add(line);
66    Date endLineTimeStamp = new Date();
    // Get the duration it took, to read a line in milliseconds.
68    long timeToReadLine = endLineTimeStamp.getTime() - startLineTimeStamp.getTime();
    if (timeData.containsKey(line.length())) {
70        timeData.get(line.length()).add(timeToReadLine);
    } else {
72        List<Long> timeList = new ArrayList<>();
        timeList.add(timeToReadLine);
74        timeData.put(line.length(), timeList);
    }
76    }
    // Sort out the summary of the run
78    Date end = new Date();
    long begin = start.getTime();
80    long fin = end.getTime();
    System.out.println("run time is " + (fin - begin) + " milliseconds");
82    return lines;
    }
84 }

```


A.0.5 Excel Writer listing

```

package workshop;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Set;

import org.apache.poi.ss.usermodel.Cell;
import org.apache.poi.xssf.usermodel.XSSFRow;
import org.apache.poi.xssf.usermodel.XSSFSheet;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;

public class ExcelWriter {

    static XSSFWorkbook workbook;

    public void getList(List<Integer> numbersOfPoints, List<Long> stoppedTimes) {
        Map<Integer, List<?>> hashmap = new HashMap<Integer, List<?>>();
        hashmap.put(0, numbersOfPoints);
        hashmap.put(1, stoppedTimes);
        System.out.println("hashmap : " + hashmap);
        Set<Integer> keyset = hashmap.keySet();
        int rownum = 0;
        int cellnum = 0;
        XSSFSheet sheet = workbook.getSheetAt(0);
        rownum = 0;
        for (Integer key : keyset) {
            List<?> nameList = hashmap.get(key);
            for (Object s : nameList) {
                XSSFRow row = sheet.getRow(rownum++);
                Cell cell = row.getCell(cellnum);
                if (null != cell) {
                    if (s instanceof Long) {
                        cell.setCellValue((Long) s);
                    }
                    if (s instanceof Integer) {
                        cell.setCellValue((Integer) s);
                    }
                }
            }
            cellnum++;
            rownum = 0;
        }
    }
}

```

```
    }  
50 }  
  
52 public static void writeExcelOutput(Map<Integer, Long> normalizedData, String FileName) throws  
    IOException {  
    workbook = new XSSFWorkbook();  
54    List<Integer> keys = new ArrayList<>();  
    List<Long> values = new ArrayList<>();  
56    for (Integer key : normalizedData.keySet()) {  
        keys.add(key);  
58        values.add(normalizedData.get(key));  
    }  
60    ExcelWriter writer = new ExcelWriter();  
    FileOutputStream out = new FileOutputStream(new File(FileName));  
62    XSSFSheet sheet = workbook.createSheet();  
    for (int i = 0; i < normalizedData.keySet().size(); i++) {  
64        XSSFRow row = sheet.createRow(i);  
        for (int j = 0; j < 2; j++)  
66            row.createCell(j);  
    }  
68    workbook.write(out);  
    out.close();  
70    InputStream inp = new FileInputStream(new File(FileName));  
    workbook = new XSSFWorkbook(inp);  
72    writer.getList(keys, values);  
    out = new FileOutputStream(new File(FileName));  
74    workbook.write(out);  
    out.close();  
76    inp.close();  
    }  
78 }
```