

1 Hammingfolge

1.1 Lösungsidee

Folgend ist die Lösungsidee für die Aufgabenstellung Hammingfolge berechnen angeführt.

Da es sich hierbei lediglich um einen einzigen Algorithmus handelt soll dieser als Klassenmethode implementiert werden. Da diese Klasse lediglich diese Klassenmethode enthalten soll, soll in dieser Klasse ein Privater Konstruktor implementiert werden um zu verhindern, dass diese Klasse instanziiert werden kann.

Da eine Hammingfolge wie folgt definiert ist:

$$1 \in H$$

$$x \in H \Rightarrow 2 * x \in H \wedge 3 * x \in H \wedge 5 * x \in H$$

wissen wir dass folgende Elemente Aufgrund dessen das $1 \in H$ gilt in der Folge vorhanden sind.

$$1 \in H \wedge 2 \in H \wedge 3 \in H \wedge 5 \in H$$

daher können wir einen Algorithmus definieren der sich wie folgt verhalten soll:

1. Instanziere eine `TreeSet<E>` und initialisiere dieses Set mit dem Element 1
2. Instanziere eine `List<E>` welches die resultierenden Werte beinhaltet wird
3. Polle und entferne das erste Element aus dem Set
4. Füge dieses Element der resultierenden Liste hinzu.
5. Berechne die nachfolgenden Hammingzahlen ($2 * polledValue \wedge 3 * polledValue \wedge 5 * polledValue$) für dieses Element
6. Füge die Berechneten Elemente dem Set hinzu
7. Wiederhole Schritt 3 solange folgendes gilt: $resultList.size(i) < n$

Für das zu verwendende Set soll eine `TreeSet<E>` Instanz verwendet werden. Es soll aber gegen `NavigableSet<E>` Interface und nicht gegen `SortedSet<E>` gearbeitet werden, da dieses Interface eine Methode namens `instance.pollFirst()` zur Verfügung stellt, die das erste Element des Set liefert und es gleichzeitig aus dem Set entfernt. Dadurch sollte das Set in seiner Größe beschränkt halten, was den Sortierungsaufwand des Set minimal halten wird.

Da `TreeSet<E>` aber auch `SortedSet<E>` implementiert sind die enthaltenen Werte implizit immer sortiert und dadurch auch die Werte in der resultierenden Liste, da die hinzugefügten Elemente immer sortiert eingefügt werden. Daher ist hier kein zusätzlicher Sortierungsaufwand nötig.

Übung 3

1.2 Source-Code

Folgend ist der Implementierte Source und Test-Source angeführt.

../src/main/java/at/fhooe/swe4/lab3/hamming/Hamming.java

```

1 package at.fhooe.swe4.lab3.hamming;
2
3 import java.math.BigInteger;
4 import java.util.ArrayList;
5 import java.util.List;
6 import java.util.NavigableSet;
7 import java.util.TreeSet;
8
9 /**
10  * Provides class methods which are used to handle hamming numbers.
11  *
12  * @author Thomas Herzog
13  */
14
15 public class Hamming {
16
17     private static final BigInteger second = BigInteger.valueOf(2);
18     private static final BigInteger three = BigInteger.valueOf(3);
19     private static final BigInteger five = BigInteger.valueOf(5);
20
21     /**
22      * Not meant to be instantiated
23      */
24     private Hamming() {
25         super();
26     }
27
28     /**
29      * Calculates the 'count' hamming numbers.
30      *
31      * @param count
32      *         the count of to calculate hamming numbers
33      * @return the sorted list holding the hamming numbers
34      */
35     public static List<BigInteger> calculateHammingNumbers(final int count) {
36         // At least one is in the hamming list
37         if (count <= 1) {
38             throw new IllegalArgumentException("The count must be at least one !!!");
39         }
40         // Avoid grow of ArrayList
41         final List<BigInteger> list = new ArrayList<BigInteger>(count);
42         // Keeps calculated elements sorted
43         final NavigableSet<BigInteger> sortedSet = new TreeSet<BigInteger>();
44         // ONE is initial value
45         sortedSet.add(BigInteger.ONE);
46         // As long as we need to calculate
47         while (list.size() != count) {
48             // Get the next hamming number and remove from set
49             final BigInteger currentValue = sortedSet.pollFirst();
50             // Add this element to result list
51             list.add(currentValue);
52             // calculate next hamming numbers
53             sortedSet.add(currentValue.multiply(second));
54             sortedSet.add(currentValue.multiply(three));
55             sortedSet.add(currentValue.multiply(five));
56         }
57         // Returned list is implicitly sorted
58         return list;
59     }
60 }

```

../src/test/java/at/fhooe/swe4/lab3/test/hamming/HammingTest.java

```

1 package at.fhooe.swe4.lab3.test.hamming;
2
3 import java.math.BigInteger;
4 import java.util.List;
5
6 import org.junit.Test;
7 import org.junit.runner.RunWith;
8 import org.junit.runners.JUnit4;
9
10 import at.fhooe.swe4.lab3.hamming.Hamming;
11
12 /**
13  * This is the test for the calculating of the hamming numbers.
14  *
15  * @author Thomas Herzog
16  */
17
18 @RunWith(JUnit4.class)
19 public class HammingTest {
20
21     @Test(expected = IllegalArgumentException.class)
22     public void test_invalid_count_negative() {
23         Hamming.calculateHammingNumbers(-1);
24     }
25
26     @Test(expected = IllegalArgumentException.class)
27     public void test_invalid_count_zero() {

```

Übung 3

```

28     Hamming.calculateHammingNumbers(0);
29 }
30
31 @Test
32 public void test_caluculation() {
33     int count = 1;
34     final int factor = 10;
35     final int repetition = 6;
36     for (int i = 0; i < repetition; i++) {
37         count *= factor;
38         final long startMillis = System.currentTimeMillis();
39         final List<BigInteger> result = Hamming.calculateHammingNumbers(count);
40         final long diffMillis = System.currentTimeMillis() - startMillis;
41         System.out.println("Spend time in millis: ");
42         System.out.println(new StringBuilder().append(diffMillis).append(" for ").
append(count).append(" hamming numbers")
.toString());
43         for (int j = 0; (j < result.size()) && (j < 20); j++) {
44             System.out.println(new StringBuilder().append(j + 1).append(": ").append(result.get(j)).toString());
45         }
46         System.out.println(" ");
47         System.out.println(" ");
48     }
49 }
50 }
51 }

```

Übung 3

1.3 Tests

Folgend sind die Tests der Aufgabenstellung Hammingfolge angeführt.

Aufgrund dessen dass JUnit verwendet wurde und die Tests mithilfe der Entwicklungsumgebung Eclipse getestet wurden ist hier nur ein Screenshot der Eclipse Umgebung angeführt.

Die Tests können in Eclipse erneut ausgeführt und dadurch auch reproduziert werden.

Die Zeitangaben der Berechnungsdauer werden bei den Tests auf die Konsole ausgegeben.

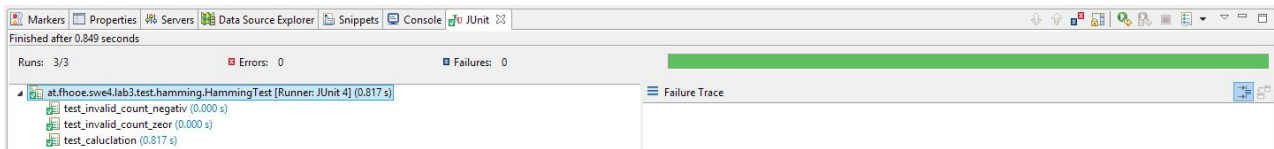


Abbildung 1: Diese Abbildung zeigt das Resultat der JUnit Tests im eclipse

```

-----
Spend time in millis: '0' for '10' hamming numbers
1: 1
2: 2
3: 3
4: 4
5: 5
6: 6
7: 8
8: 9
9: 10
10: 12
-----
-----
Spend time in millis: '0' for '100' hamming numbers
1: 1
2: 2
3: 3
4: 4
5: 5
6: 6
7: 8
8: 9
9: 10
10: 12
11: 15
12: 16
13: 18
14: 20
15: 24
16: 25
17: 27
18: 30
19: 32
20: 36
-----
-----

```

Abbildung 2: Diese Abbildung zeigt die Berechnungszeiten für 10 und 100 Hammingzahlen

Übung 3

```
-----  
-----  
Spend time in millis: '8' for '1000' hamming numbers  
1: 1  
2: 2  
3: 3  
4: 4  
5: 5  
6: 6  
7: 8  
8: 9  
9: 10  
10: 12  
11: 15  
12: 16  
13: 18  
14: 20  
15: 24  
16: 25  
17: 27  
18: 30  
19: 32  
20: 36  
-----  
-----  
-----  
Spend time in millis: '16' for '10000' hamming numbers  
1: 1  
2: 2  
3: 3  
4: 4  
5: 5  
6: 6  
7: 8  
8: 9  
9: 10  
10: 12  
11: 15  
12: 16  
13: 18  
14: 20  
15: 24  
16: 25  
17: 27  
18: 30  
19: 32  
20: 36  
-----  
-----
```

Abbildung 3: Diese Abbildung zeigt die Berechnungszeiten für 1000 und 10.000 Hammingzahlen

Übung 3

```
-----  
Spend time in millis: '72' for '100000' hamming numbers
```

```
1: 1  
2: 2  
3: 3  
4: 4  
5: 5  
6: 6  
7: 8  
8: 9  
9: 10  
10: 12  
11: 15  
12: 16  
13: 18  
14: 20  
15: 24  
16: 25  
17: 27  
18: 30  
19: 32  
20: 36  
-----  
-----  
-----
```

```
-----  
Spend time in millis: '712' for '1000000' hamming numbers
```

```
1: 1  
2: 2  
3: 3  
4: 4  
5: 5  
6: 6  
7: 8  
8: 9  
9: 10  
10: 12  
11: 15  
12: 16  
13: 18  
14: 20  
15: 24  
16: 25  
17: 27  
18: 30  
19: 32  
20: 36  
-----  
-----
```

Abbildung 4: Diese Abbildung zeigt die Berechnungszeiten für 100.000 und 1.000.000 Hammingzahlen

2 Sortieralgorithmen

2.1 Lösungsidee (Allgemein)

Folgend sind die Lösungsideen der Sortieralgorithmen HeapSorter und QuickSorter angeführt.

Da beide Algorithmen denselben output liefern sollen, soll hier ein Interface spezifiziert werden welches die Funktionalität bzw. die zu implementierenden Methoden Signaturen vorgibt. Die Aufgabenstellung verlangt zwar nur das Sortieren auf Integer Felder, jedoch sollen die Algorithmen so implementiert werden, dass sie auf Typen, die das Interface Comparable<E> implementieren. Daher muss das Interface folgende Signatur vorweisen. `public Sorter<E extends Comparable<E>> {...}`

Übung 3

2.1.1 Source Code Allgemein

Folgend sind die allgemeinen Sources der Sortieralgorithmen angeführt.

../src/main/java/at/fhooe/swe4/lab3/sort/api/Sorter.java

```

1 package at.fhooe.swe4.lab3.sort.api;
2
3 import java.util.List;
4
5 import at.fhooe.swe4.lab3.stat.api.StatisticsProvider;
6
7 /**
8  * This interface specifies the sorter functionalities.
9  *
10  * @author Thomas Herzog
11  *
12  * @param <V>
13  *         the values type of the collections or array elements
14  */
15 public interface Sorter<V extends Comparable<V>> {
16     /**
17      * This enumeration specifies the sort order for a heap sort instance.
18      *
19      * @author Thomas Herzog
20      *
21      */
22     public static enum SortType {
23         /**
24          * Will result in an ascending ordered result
25          */
26         DESCENDING,
27         /**
28          * Will result in an descending ordered result
29          */
30         ASCENDING;
31
32         /**
33          * Compares the two comparable instances.
34          * <ul>
35          * <li>
36          * {@link SortType#DESCENDING} performs an x < 0 comparision</li>
37          * <li>{@link SortType#ASCENDING} performs an x > 0 comparision</li>
38          * </ul>
39          *
40          * @param left
41          *         the instance which invokes the compareTo method
42          * @param right
43          *         the parameter for lefts compareTo method invocation
44          * @return the proper result for the specified heap type
45          */
46         public <T extends Comparable<T>> boolean compare(T left, T right) {
47             switch (this) {
48                 case DESCENDING:
49                     return left.compareTo(right) > 0;
50                 case ASCENDING:
51                     return left.compareTo(right) <= 0;
52                 default:
53                     throw new IllegalStateException("This enum is not handled here but should. enum=" + this.name());
54             }
55         }
56     }
57
58     /**
59      * Sorts the given array.
60      *
61      * @param array
62      *         the array to be sorted
63      * @param sorterType
64      *         the type of the sorting
65      * @return the sorted array
66      * @see SortType
67      * @throws IllegalArgumentException
68      *         if the array is null, or the {@link SortType} is null
69      */
70     public V[] sort(V[] array, SortType sorterType);
71
72     /**
73      * Sorts the given list5.
74      *
75      * @param list
76      *         the list to be sorted
77      * @param sorterType
78      *         the type of the sorting
79      * @return the sorted array
80      * @see SortType
81      * @throws IllegalArgumentException
82      *         if the list is null, or the {@link SortType} is null
83      */
84     public List<V> sort(List<V> list, SortType sorterType);
85
86     /**
87      * Gets the statistics of the current instance
88      *
89      * @return the current statistics
90      */
91     public StatisticsProvider getStatistitcs();
92 }

```


2.2 Lösungsidee (Statistics)

Aufgrund dessen dass die Sortieralgorithmen mit Code Statistics versehen werden sollen, sollen Klassen implementiert werden, die es erlauben die verlangten Statistics zu ermitteln und auch einen Report dieser zu erstellen.

Hierbei soll diese Code Statistics wie folgt aufgeteilt werden:

1. **StatisticsProvider:** Das Interface welches die Spezifikation für den Code Statistik Provider enthalten soll.
Die Implementierung soll es ermöglichen mehrere Statistik Kontexte zu verwalten.
2. **StatisticContext:** Die Klasse, welche einen Statistik Kontext darstellen soll.
Dieser Kontext soll es ermöglichen mehrere CodeStatistic Instanzen pro Kontext zu verwalten.
3. **CodeStatistic:** Die Klasse, die die Code Statistik Informationen (swap, compare counts) halten soll
4. **DefaultStatisticProviderImpl:** Die default Implementierung des Interface StatisticProvider, welches die Funktionalitäten implementiert soll.

Alle Klassen sollen die `instance.toString()` Methode überschreiben und jeweils ihre beinhaltenden Informationen als String zurückliefern, wobei ein Parent bzw. die Klasse die Instanzen einer anderen verwaltet and deren `child.toString()` zu delegieren.

Übung 3

2.2.1 Source Code

Folgend ist der Source der Statistik Implementierungen und Interfaces angeführt.

../src/main/java/at/fhooe/swe4/lab3/stat/api/StatisticsProvider.java

```

1 package at.fhooe.swe4.lab3.stat.api;
2
3 import at.fhooe.swe4.lab3.stat.StatisticContext;
4
5
6 /**
7  * This interfaces specifies the functionalities of an statistics provider
8  * instance.
9  *
10 * @author Thomas Herzog
11 *
12 */
13 public interface StatisticsProvider {
14
15     /**
16      * Initializes a new context where code statistics are placed
17      *
18      * @param key
19      *         the key of the context. If present in backed set then existing
20      *         statistics will be lost
21      * @return the current instance
22      */
23     public StatisticsProvider initContext(String key);
24
25     /**
26      * Ends the current context by setting its end date and by setting the
27      * current context null.
28      *
29      * @return the current instance
30      */
31     public StatisticsProvider endContext();
32
33     /**
34      * Removes an existing context. Does nothing if key not found.
35      *
36      * @param key
37      *         the key of the context to be removed
38      * @return the current instance
39      */
40     public StatisticsProvider removeContext(String key);
41
42     /**
43      * Takes and statistic provider. If a provider is already present with the
44      * given key then the existing provider will be lost.
45      *
46      * @param key
47      *         the key to map this provider to
48      * @param provider
49      *         the provider to be taken over
50      * @return the current instance
51      */
52     public StatisticsProvider takeOver(String key, StatisticsProvider provider);
53
54     /**
55      * Gets the current active {@link StatisticContext}.
56      *
57      * @return the current active statistics context, can be null
58      */
59     public StatisticContext getCtx();
60 }

```

../src/main/java/at/fhooe/swe4/lab3/stat/StatisticContext.java

```

1 package at.fhooe.swe4.lab3.stat;
2
3 import java.util.Calendar;
4 import java.util.Collections;
5 import java.util.Comparator;
6 import java.util.Set;
7 import java.util.SortedSet;
8 import java.util.TreeSet;
9
10 import org.apache.commons.collections4.CollectionUtils;
11 import org.apache.commons.collections4.Predicate;
12 import org.apache.commons.lang3.StringUtils;
13 import org.apache.commons.lang3.time.DateFormatUtils;
14
15 /**
16  * This class represents a statistic context which is used for statistic code
17  * analysis.
18  *
19  * @author Thomas Herzog
20  *
21 */
22 public class StatisticContext {
23     private final String key;
24     public Calendar startCalendar;
25     public Calendar endCalendar;
26     public final SortedSet<CodeStatistics> statisticsSet;
27

```

Übung 3

```

28  /**
29   * Default constructor which creates a context identified by the given
30   * unique key.
31   *
32   * @param key
33   *         the unique key for the created context
34   * @throws IllegalArgumentException
35   *         if the key is either null or an empty string
36   */
37  public StatisticContext(final String key) {
38      super();
39      if (StringUtils.isEmpty(key)) {
40          throw new IllegalArgumentException("StatisticContext instance must be identified by an unique string
41          key");
42      }
43      this.key = key;
44      this.statisticsSet = new TreeSet<CodeStatistics>(new Comparator<CodeStatistics>() {
45          @Override
46          public int compare(CodeStatistics left, CodeStatistics right) {
47              return left.getKey().compareTo(right.getKey());
48          }
49      });
50
51  /**
52   * Gets the start calendar instance
53   *
54   * @return the start calendar
55   */
56  public Calendar getStartCalendar() {
57      return startCalendar;
58  }
59
60  /**
61   * Sets the start calendar instance
62   *
63   * @param startCalendar
64   *         the start calendar instance
65   */
66  public void setStartCalendar(Calendar startCalendar) {
67      this.startCalendar = startCalendar;
68  }
69
70  /**
71   * Gets the end calendar instance
72   *
73   * @return the end calendar instance
74   */
75  public Calendar getEndCalendar() {
76      return endCalendar;
77  }
78
79  /**
80   * Sets the end calendar instance
81   *
82   * @param endCalendar
83   *         the end calendar instance
84   */
85  public void setEndCalendar(Calendar endCalendar) {
86      this.endCalendar = endCalendar;
87  }
88
89  /**
90   * Gets the statistic set which contains the code statistics of the current
91   * context.
92   *
93   * @return the code statistics of the current context
94   */
95  public Set<CodeStatistics> getStatisticsSet() {
96      return Collections.unmodifiableSet(statisticsSet);
97  }
98
99  /**
100   * Adds a code statistic instance to the backed set.
101   *
102   * @param statistics
103   *         the code statistics to be added to the set
104   * @return the current instance
105   */
106  public StatisticContext addStatistics(final CodeStatistics statistics) {
107      statisticsSet.add(statistics);
108      return this;
109  }
110
111  /**
112   * @param key
113   *         the key of the code statistic instance.
114   * @return the code statistic instance, null otherwise
115   * @see StatisticContext#byKey(String, boolean)
116   */
117  public CodeStatistics byKey(final String key) {
118      return byKey(key, Boolean.FALSE);
119  }
120
121  /**
122   * Gets a code statistics identified by the given key.
123   *
124   * @param statKey
125   *         the key of the code statistic instance.

```

Übung 3

```

126  * @param newIfNot
127  *         true if a new instance should be created if not found in the
128  *         backed set
129  * @return the code statistic instance, null otherwise
130  */
131 public CodeStatistics byKey(final String statKey, boolean newIfNot) {
132     CodeStatistics stat = CollectionUtils.find(statisticsSet, new Predicate<CodeStatistics>() {
133         @Override
134         public boolean evaluate(CodeStatistics object) {
135             return object.getKey().equals(statKey.trim().toLowerCase());
136         }
137     });
138     return (stat != null) ? stat : newStatistic(statKey);
139 }
140
141 /**
142  * Creates a new code statistic instance
143  *
144  * @param key
145  *         the key for the code statistic instance
146  * @return the new code statistic instance
147  */
148 public CodeStatistics newStatistic(final String key) {
149     final CodeStatistics stat = new CodeStatistics(key);
150     statisticsSet.add(stat);
151     return stat;
152 }
153
154 /**
155  * @return the formatted start calendar string representation
156  * @see StatisticContext#formatDate(Calendar)
157  */
158 public String formattedStartDate() {
159     return formatDate(startCalendar);
160 }
161
162 /**
163  * @return the formatted end calendar string representation
164  * @see StatisticContext#formatDate(Calendar)
165  */
166 public String formattedEndDate() {
167     return formatDate(endCalendar);
168 }
169
170 /**
171  * Creates a string representation of the given calendar instance
172  *
173  * @param cal
174  *         the calendar instance to be formatted
175  * @return the formatted calendar string
176  */
177 private String formatDate(final Calendar cal) {
178     return DateFormatUtils.format(cal, "HH:mm:ss:SSS");
179 }
180
181 /**
182  * @return the key of this statistic context.
183  */
184 public String getKey() {
185     return key;
186 }
187
188 /**
189  * Prints the statistic context and its code statistics
190  */
191 @Override
192 public String toString() {
193     final String ln = System.getProperty("line.separator");
194     final StringBuilder sb = new StringBuilder(500);
195     sb.append("#####").append(ln);
196     sb.append("## statistic-context-key: ").append(key).append(ln);
197     sb.append("## start-date: ").append(formattedStartDate()).append(ln);
198     sb.append("## end-date: ").append(formattedEndDate()).append(ln);
199     sb.append("#####").append(ln);
200     for (CodeStatistics statistics : statisticsSet) {
201         sb.append(statistics.toString()).append(ln);
202     }
203     sb.append("#####").append(ln);
204     return sb.toString();
205 }
206 }

```

../src/main/java/at/fhooe/swe4/lab3/stat/CodeStatistics.java

```

1 package at.fhooe.swe4.lab3.stat;
2
3 import org.apache.commons.lang3.StringUtils;
4
5 /**
6  * This is model which holds the code statistic data.
7  *
8  * @author Thomas Herzog
9  */
10
11 public class CodeStatistics {

```

Übung 3

```

12
13 private int comparision;
14 private int swaps;
15
16 private final String key;
17
18 /**
19  *
20  * @param key
21  * the key for this statistics data.
22  * @throws IllegalArgumentException
23  * if the key is either null or an empty string
24  */
25 public CodeStatistics(final String key) {
26     super();
27     if (StringUtils.isEmpty(key)) {
28         throw new IllegalArgumentException("Statistic instance must be identified by an unique string key");
29     }
30     this.key = key.trim().toLowerCase();
31     clear();
32 }
33
34 /**
35  * The key of this instance
36  *
37  * @return the instance key
38  */
39 public String getKey() {
40     return key;
41 }
42
43 /**
44  * Increase the comparison counter.
45  *
46  * @return the current instance
47  */
48 public CodeStatistics incIf() {
49     comparision++;
50     return this;
51 }
52
53 /**
54  * Increases the swap counter
55  *
56  * @return the current instance
57  */
58 public CodeStatistics incSwap() {
59     swaps++;
60     return this;
61 }
62
63 /**
64  * Clears the code statistic by setting all counters to '0'
65  *
66  * @return the current instance
67  */
68 public CodeStatistics clear() {
69     comparision = 0;
70     swaps = 0;
71     return this;
72 }
73
74 // Sorter statistics
75 /**
76  * @return the comparison counter
77  */
78 public int getComparision() {
79     return comparision;
80 }
81
82 /**
83  * @return the swap counter
84  */
85 public int getSwaps() {
86     return swaps;
87 }
88
89 public int hashCode() {
90     final int prime = 31;
91     int result = 1;
92     result = prime * result + ((key == null) ? 0 : key.hashCode());
93     return result;
94 }
95
96 public boolean equals(Object obj) {
97     if (this == obj)
98         return true;
99     if (obj == null)
100         return false;
101     if (getClass() != obj.getClass())
102         return false;
103     CodeStatistics other = (CodeStatistics) obj;
104     if (key == null) {
105         if (other.key != null)
106             return false;
107     } else if (!key.equals(other.key))
108         return false;
109     return true;
110 }

```

Übung 3

```

111
112 @Override
113 public String toString() {
114     final String ln = System.getProperty("line.separator");
115     final StringBuilder sb = new StringBuilder();
116     sb.append("## statistic-key: ").append(key).append(ln);
117     sb.append(String.format("## %15s %15s", "comparisions", "swaps")).append(ln);
118     sb.append(String.format("## %15s %15s", comparision, swaps));
119     return sb.toString();
120 }
121 }

```

../src/main/java/at/fhooe/swe4/lab3/stat/DefaultStatisticsProviderImpl.java

```

1 package at.fhooe.swe4.lab3.stat;
2
3 import java.util.Calendar;
4 import java.util.Comparator;
5 import java.util.HashMap;
6 import java.util.Map;
7 import java.util.Map.Entry;
8 import java.util.SortedSet;
9 import java.util.TreeSet;
10
11 import org.apache.commons.collections4.CollectionUtils;
12 import org.apache.commons.collections4.Predicate;
13 import org.apache.commons.lang3.StringUtils;
14
15 import at.fhooe.swe4.lab3.stat.api.StatisticsProvider;
16
17 /**
18  * This is the default statistics provider implementation.
19  *
20  * @author Thomas Herzog
21  */
22
23 public class DefaultStatisticsProviderImpl implements StatisticsProvider {
24
25     private StatisticContext ctx = null;
26     private final Map<String, StatisticsProvider> providers;
27     private final SortedSet<StatisticContext> statContextSet = new TreeSet<StatisticContext>(new Comparator<
28         StatisticContext>() {
29         @Override
30         public int compare(StatisticContext left, StatisticContext right) {
31             return left.getStartCalendar().compareTo(right.getStartCalendar());
32         }
33     });
34
35     /**
36      * Default constructor which creates an context with the given key
37      *
38      * @param contextKey
39      *         the key for the initial context
40      */
41     public DefaultStatisticsProviderImpl() {
42         super();
43         this.providers = new HashMap<String, StatisticsProvider>();
44     }
45
46     @Override
47     public StatisticsProvider initContext(final String contextKey) {
48         if (ctx != null) {
49             endContext();
50         }
51         if (StringUtils.isEmpty(contextKey)) {
52             throw new IllegalArgumentException("Context key must be given fopr context");
53         }
54         ctx = new StatisticContext(contextKey.trim().toLowerCase());
55         ctx.setStartCalendar(Calendar.getInstance());
56         statContextSet.add(ctx);
57         return this;
58     }
59
60     @Override
61     public StatisticsProvider endContext() {
62         if (ctx != null) {
63             ctx.setEndCalendar(Calendar.getInstance());
64             ctx = null;
65         }
66         return this;
67     }
68
69     @Override
70     public StatisticsProvider removeContext(final String ctxDelKey) {
71         final StatisticContext ctxDel = CollectionUtils.find(statContextSet, new Predicate<StatisticContext>() {
72             @Override
73             public boolean evaluate(StatisticContext object) {
74                 return object.getKey().equals(ctxDelKey.trim().toLowerCase());
75             }
76         });
77         if (ctxDel != null) {
78             statContextSet.remove(ctxDel);
79         }
80         return this;
81     }
82
83     @Override
84     public StatisticsProvider takeOver(final String key, final StatisticsProvider provider) {

```

Übung 3

```

84     if (StringUtils.isEmpty(key) || (provider == null)) {
85         throw new IllegalArgumentException("Key and provider must be given");
86     }
87     this.providers.put(key, provider);
88     return this;
89 }
90
91 @Override
92 public StatisticContext getCtx() {
93     return ctx;
94 }
95
96 @Override
97 public String toString() {
98     final String ln = System.getProperty("line.separator");
99     final StringBuilder sb = new StringBuilder(500);
100    sb.append("#####").append(ln);
101    ;
102    sb.append("## statistic-context-provider").append(ln).append("##").append(ln);
103    sb.append("## statistics-contexts-count:").append(statContextSet.size()).append(ln).append("##").append(
104        ln);
105    for (StatisticContext ctx : statContextSet) {
106        sb.append(ctx.toString()).append(ln);
107    }
108    // Other providers
109    if (!providers.isEmpty()) {
110        sb.append("Managed providers:").append(ln);
111        for (Entry<String, StatisticsProvider> entry : providers.entrySet()) {
112            sb.append("Key:").append(entry.getKey()).append(ln);
113            sb.append(entry.getValue().toString()).append(ln);
114        }
115        sb.append("#####").append(
116            ln);
117    }
118    return sb.toString();
119 }

```

Übung 3

2.3 HeapSorter

Folgend ist die Lösungsidee für die HeapSorter Implementierung angeführt.

Da hierbei eine Heap Implementierung von Nöten ist und diese aber auch anderweitig verwendet werden könnte soll ein Heap Implementiert werden, der unabhängig von einem HeapSorter verwendet werden kann. Da wir auch hier generisch bleiben wollen und es auch möglich sein soll eine Heap Implementierung mit einem anderen Container zu implementieren (Bsp.: `ArrayList<E>`, `T[]`, usw.) soll ein Interface spezifiziert werden, welches die Funktionalitäten eines Heap spezifiziert. Es soll folgende Signatur haben `public Heap<E extends Comparable<E>> {...}`

Des Weiteren soll eine Enumeration spezifiziert werden, die es erlaubt zu definieren, ob der Heap ein upheap oder downheap sein soll, also ob der root das höchste oder kleinste Element darstellt.

Ansonsten soll der Heap wie bekannt implementiert werden.

2.3.1 Source Code

Folgend ist der Source der Interfaces und Implementierungen für Heap und Heap Sorter angeführt.

../src/main/java/at/fhooe/swe4/lab3/sort/api/Heap.java

```
1 package at.fhooe.swe4.lab3.sort.api;
2
3 import java.util.Collection;
4 import java.util.List;
5
6 import at.fhooe.swe4.lab3.stat.api.StatisticsProvider;
7
8 /**
9  * This interface specifies the heap functionalities.
10  *
11  * @author Thomas Herzog
12  *
13  * @param <V>
14  *         the value type of the elements in the heap
15  */
16 public interface Heap<V extends Comparable<V>> {
17
18     /**
19      * This enumeration specifies the supported heap types
20      *
21      * @author Thomas Herog
22      *
23      */
24     public static enum HeapType {
25         /**
26          * Will result an ascending ordered heap
27          */
28         MAX_HEAP, /**
29          * Will result an descending ordered heap
30          */
31         MIN_HEAP;
32
33         /**
34          * Compares the two comparable instances.
35          * <ul>
36          * <li>
37          * {@link HeapType#MIN_HEAP} performs an x < 0 comparision</li>
38          * <li>{@link HeapType#MIN_HEAP} performs an x > 0 comparision</li>
39          * </ul>
40          *
41          * @param left
42          *         the instance which invokes the comparesTo method
43          * @param right
44          *         the parameter for lefts compareTomethod invocation
45          * @return the proper result for the specified heap type
46          */
47         public <T extends Comparable<T>> boolean compare(T left, T right) {
48             switch (this) {
49                 case MAX_HEAP:
50                     return left.compareTo(right) < 0;
51                 case MIN_HEAP:
52                     return left.compareTo(right) > 0;
53                 default:
54                     throw new IllegalStateException("This enum is not handled here but should. enum=" + this.name());
55             }
56         }
57     }
58
59     /**
60      * Initializes this heap with the given array of elements.
61      *
62      * @param originalArrayValues
63      *         the values to build an heap structure from
64      * @param sortType
65      *         the type of how the elements should be
```


Übung 3

```

66  */
67  public void init(V[] originalArrayValues, HeapType sortType);
68
69  /**
70   * Initializes this heap with the given collection which provides the
71   * elements.
72   *
73   * @param originalArrayValues
74   *       the values to build an heap structure from
75   * @param sortType
76   *       the type of how the elements should be
77   */
78  public void init(Collection<V> originalIterableValues, HeapType sortType);
79
80  /**
81   * Puts an element on the heap and keeps heap type specified order.
82   *
83   * @param value
84   *       the element to be put on the heap
85   */
86  public void enqueue(V value);
87
88  /**
89   * Gets the top element of the heap
90   *
91   * @return the top element
92   */
93  public V dequeue();
94
95  /**
96   * Converts the heap to a flat list which represents the backed tree
97   * structure.
98   *
99   * @return the list representing the heap. Will be a new instance
100  */
101  public List<V> toList();
102
103  /**
104   * Converts the heap to an flat array which represents the bakced trees
105   * structure
106   *
107   * @return the array representing the heap
108   */
109  public V[] toArray();
110
111  /**
112   * Answers the question if the heap has another element
113   *
114   * @return true if there is still an element left on the heap
115   */
116  public boolean hasNext();
117
118  /**
119   * Returns the current size of the heap.
120   *
121   * @return the heap element size
122   */
123  public int size();
124
125  /**
126   * Gets the statistics of the current instance
127   *
128   * @return the current statistics
129   */
130  public StatisticsProvider getStatistitcs();
131 }

```

../src/main/java/at/fhooe/swe4/lab3/sort/heap/impl/HeapArrayListImpl.java

```

1  package at.fhooe.swe4.lab3.sort.heap.impl;
2
3  import java.util.ArrayList;
4  import java.util.Collection;
5  import java.util.Iterator;
6  import java.util.List;
7
8  import at.fhooe.swe4.lab3.sort.api.Heap;
9  import at.fhooe.swe4.lab3.stat.CodeStatistics;
10 import at.fhooe.swe4.lab3.stat.DefaultStatisticsProviderImpl;
11 import at.fhooe.swe4.lab3.stat.api.StatisticsProvider;
12
13 /**
14  * This is the ArrayList implementation of the heap.
15  *
16  * @author Thomas Herzog
17  *
18  * @param <V>
19  *       the value type of the heap managed elements
20  */
21 public class HeapArrayListImpl<V extends Comparable<V>> implements Heap<V> {
22
23     public HeapType heapType;
24     public List<V> container = new ArrayList<V>();
25
26     public StatisticsProvider statProvider = new DefaultStatisticsProviderImpl();
27
28     /**
29      * Empty constructor

```

Übung 3

```

30  */
31  public HeapArrayListImpl() {
32      super();
33  }
34
35  /**
36   * Initializes the heap with the given array
37   *
38   * @param array
39   *       the array providing the elements for the heap
40   * @param heapType
41   *       the type of the heap
42   * @see HeapType
43   */
44  public HeapArrayListImpl(final V[] array, final HeapType heapType) {
45      super();
46      init(array, heapType);
47  }
48
49  /**
50   * Initializes the heap with the given collection
51   *
52   * @param list
53   *       the collection providing the elements for the heap
54   * @param heapType
55   *       the type of the heap
56   * @see HeapType
57   */
58  public HeapArrayListImpl(final Collection<V> list, final HeapType heapType) {
59      super();
60      init(list, heapType);
61  }
62
63  @Override
64  public void init(final V[] originalArrayValues, final HeapType heapType) {
65      this.heapType = heapType;
66      int size = ((originalArrayValues == null) || (originalArrayValues.length == 0)) ? 0 : originalArrayValues
        .length;
67      statProvider.initContext(new StringBuilder(this.getClass().getSimpleName()).append(" elements").append(
        size).append("]").toString());
68      if (size > 0) {
69          container = new ArrayList<V>(size);
70          final CodeStatistics stat = statProvider.getCtx().newStatistic("init(array)");
71          for (V value : originalArrayValues) {
72              enqueue(value);
73          }
74      } else {
75          container = new ArrayList<V>(0);
76      }
77  }
78
79  @Override
80  public void init(final Collection<V> originalIterableValues, final HeapType heapType) {
81      this.heapType = heapType;
82      final int size = (originalIterableValues == null) ? 0 : originalIterableValues.size();
83      statProvider.initContext(new StringBuilder(this.getClass().getSimpleName()).append(" elements").append(
        size).append("]").toString());
84      if (size > 0) {
85          container = new ArrayList<V>(size);
86          final Iterator<V> it = originalIterableValues.iterator();
87          while (it.hasNext()) {
88              enqueue(it.next());
89          }
90      } else {
91          container = new ArrayList<V>(0);
92      }
93  }
94
95  @Override
96  public void enqueue(final V value) {
97      container.add(value);
98      upHeap(container);
99  }
100
101  @Override
102  public V dequeue() {
103      final V value = container.get(0);
104      container.set(0, container.get(container.size() - 1));
105      downHeap(container);
106      container.remove(container.size() - 1);
107      return value;
108  }
109
110  @Override
111  public boolean hasNext() {
112      return container.size() > 0;
113  }
114
115  @Override
116  public int size() {
117      return container.size();
118  }
119
120  @Override
121  public List<V> toList() {
122      return new ArrayList<V>(container);
123  }
124
125  @Override

```

Übung 3

```

126 @SuppressWarnings("unchecked")
127 public V[] toArray() {
128     return (V[]) container.toArray();
129 }
130
131 @Override
132 public StatisticsProvider getStatistics() {
133     return statProvider.endContext();
134 }
135
136 // Private heap methods
137 /**
138  * Performs an up heap on the given heap represented by the given list
139  *
140  * @param container
141  *     the list representing the heap
142  */
143 private void upHeap(final List<V> container) {
144     final CodeStatistics stat = statProvider.getCtx().byKey("upHeap()", Boolean.TRUE);
145
146     int i = container.size() - 1;
147     V tmp = container.get(i);
148     while ((i != 0) && (heapType.compare(container.get(parent(i)), tmp))) {
149         stat.incIf().incSwap();
150         container.set(i, container.get(parent(i)));
151         i = parent(i);
152     }
153     container.set(i, tmp);
154 }
155
156 /**
157  * Performs an down heap on the given heap represented by the given list
158  *
159  * @param container
160  *     the list representing the heap
161  */
162 private void downHeap(final List<V> container) {
163     final CodeStatistics stat = statProvider.getCtx().byKey("downHeap()", Boolean.TRUE);
164     int idx = 0;
165     int largeIdx;
166     V tmp = container.get(0);
167     while (idx < (container.size() / 2)) {
168         int leftIdx = left(idx);
169         int rightIdx = right(idx);
170         stat.incIf();
171         if ((rightIdx < container.size()) && (heapType.compare(container.get(leftIdx), container.get(rightIdx))
172         )) {
173             largeIdx = rightIdx;
174         } else {
175             largeIdx = leftIdx;
176         }
177         stat.incIf();
178         if (!heapType.compare(tmp, container.get(largeIdx))) {
179             break;
180         }
181         stat.incSwap();
182         container.set(idx, container.get(largeIdx));
183         idx = largeIdx;
184     }
185     container.set(idx, tmp);
186 }
187
188 // Private helper
189 /**
190  * Gets the parent index of the element on index i
191  *
192  * @param i
193  *     the index to get its parent index
194  * @return the parent index
195  */
196 private static int parent(final int i) {
197     return (i - 1) / 2;
198 }
199
200 /**
201  * Gets the left neighbor index of the element on index i
202  *
203  * @param i
204  *     the index to get its left neighbor index
205  * @return the left neighbor index
206  */
207 private static int left(final int i) {
208     return (i * 2) + 1;
209 }
210
211 /**
212  * Gets the right neighbor index of the element on index i
213  *
214  * @param i
215  *     the index to get its right neighbor index
216  * @return the right neighbor index
217  */
218 private static int right(final int i) {
219     return (i * 2) + 2;
220 }
221
222 @Override
223 public String toString() {
224     final int new_line_count = 10;

```

Übung 3

```

224     final StringBuilder sb = new StringBuilder();
225     sb.append(this.getClass().getName()).append(" [size=").append(container.size()).append("]\n");
226     sb.append("idx[0 - ").append(new_line_count).append("]: ");
227     for (int i = 0; i < container.size(); i++) {
228         sb.append(container.get(i));
229         if ((i + 1) < container.size()) {
230             sb.append(", ");
231         }
232         if ((i > 0) && (i % new_line_count == 0)) {
233             final int idxEnd = ((i + new_line_count) < container.size()) ? (i + new_line_count) : (container.size() - 1);
234             sb.append(System.getProperty("line.separator"));
235             sb.append("idx[").append(i + 1).append(" - ").append(idxEnd).append("]: ");
236         }
237     }
238     return sb.toString();
239 }
240 }

```

../src/main/java/at/fhooe/swe4/lab3/sort/heap/impl/HeapSorter.java

```

1 package at.fhooe.swe4.lab3.sort.heap.impl;
2
3 import java.util.ArrayList;
4 import java.util.Arrays;
5 import java.util.List;
6
7 import at.fhooe.swe4.lab3.sort.api.Heap;
8 import at.fhooe.swe4.lab3.sort.api.Heap.HeapType;
9 import at.fhooe.swe4.lab3.sort.api.Sorter;
10 import at.fhooe.swe4.lab3.stat.api.StatisticsProvider;
11
12 /**
13  * This is the heap sorter implementation of the Sorter interface.
14  *
15  * @author Thomas Herzog
16  *
17  * @param <V>
18  *         the values type of the to sort array or collection managed
19  *         elements
20  */
21 public class HeapSorter<V> extends Comparable<V>> implements Sorter<V> {
22
23     private final Heap<V> heap = new HeapArrayListImpl<V>();
24
25     public HeapSorter() {
26         super();
27     }
28
29     @SuppressWarnings("unchecked")
30     @Override
31     public V[] sort(final V[] array, final SortType sorterType) {
32         if (array == null) {
33             throw new IllegalArgumentException("Cannot sort empty array");
34         }
35         return (array.length == 0) ? array : ((V[]) sort(Arrays.asList(array), sorterType).toArray());
36     }
37
38     @Override
39     public List<V> sort(final List<V> list, final SortType sorterType) {
40         if (sorterType == null) {
41             throw new IllegalArgumentException("SorterType not defined");
42         }
43         if (list == null) {
44             throw new IllegalArgumentException("Cannot sort null list");
45         }
46         heap.init(list, convertToHeapType(sorterType));
47         final List<V> result = new ArrayList<V>();
48         while (heap.hasNext()) {
49             result.add(heap.dequeue());
50         }
51         return result;
52     }
53
54     @Override
55     public StatisticsProvider getStatistics() {
56         return heap.getStatistics();
57     }
58
59     /**
60      * Converts the sorter type to the corresponding heap type.
61      *
62      * @param sorterType
63      *         the sorter type to be converted
64      * @return the corresponding heap type
65      * @throws IllegalArgumentException
66      *         if the sorter type cannot be mapped to a corresponding heap
67      *         type
68      */
69     private HeapType convertToHeapType(final SortType sorterType) {
70         switch (sorterType) {
71             case ASCENDING:
72                 return HeapType.MIN_HEAP;
73             case DESCENDING:
74                 return HeapType.MAX_HEAP;
75             default:
76                 throw new IllegalArgumentException("SortType cannot be mapped to corresponding HeapType !!!");
77         }
78     }
79 }

```

Übung 3

78 }
79 }

Übung 3

2.3.2 Tests

Folgend sind die Tests für die HeapSorter Implementierung angeführt.

Diese Tests können einfach in einem eclipse erneut ausgeführt und reproduziert werden, obwohl anzumerken ist, dass die Zeiten abhängig von den zu Verfügung stehenden Ressourcen abhängig sind.

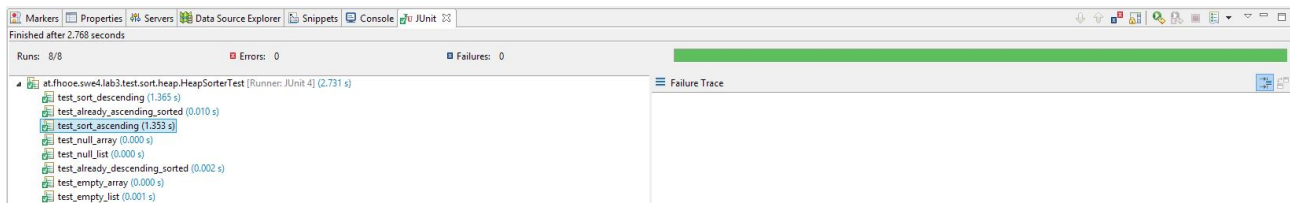


Abbildung 5: Diese Abbildung zeigt das Resultat der JUnit Tests in eclipse

Übung 3

```

test_sort_descending()
#####
## statistic-context-provider
##
## statistics-contexts-count:6
##
#####
## statistic-context-key: heaparraylistimpl elements[10]
## start-date: 23:50:18:031
## end-date: 23:50:18:046
#####
## statistic-key: downheap()
##      comparisions      swaps
##          34          12
#####
## statistic-key: upheap()
##      comparisions      swaps
##          6           6
#####

#####
## statistic-context-key: heaparraylistimpl elements[100]
## start-date: 23:50:18:046
## end-date: 23:50:18:048
#####
## statistic-key: downheap()
##      comparisions      swaps
##         866          411
#####
## statistic-key: upheap()
##      comparisions      swaps
##         105          105
#####

#####
## statistic-context-key: heaparraylistimpl elements[1000]
## start-date: 23:50:18:048
## end-date: 23:50:18:061
#####
## statistic-key: downheap()
##      comparisions      swaps
##       15058         7342
#####
## statistic-key: upheap()
##      comparisions      swaps
##        1280         1280
#####

```

Abbildung 6: Diese Abbildung zeigt die Statistiken für das Sortieren von 10, 100, 1.000 Elementen

Übung 3

```

#####
## statistic-context-key: heaparraylistimpl elements[10000]
## start-date: 23:50:18:061
## end-date: 23:50:18:113
#####
## statistic-key: downheap()
##   comparisions      swaps
##       216546          106733
#####
## statistic-key: upheap()
##   comparisions      swaps
##       12748           12748
#####

#####
## statistic-context-key: heaparraylistimpl elements[100000]
## start-date: 23:50:18:114
## end-date: 23:50:18:297
#####
## statistic-key: downheap()
##   comparisions      swaps
##       2830276         1399405
#####
## statistic-key: upheap()
##   comparisions      swaps
##       127823          127823
#####

#####
## statistic-context-key: heaparraylistimpl elements[1000000]
## start-date: 23:50:18:297
## end-date: 23:50:19:338
#####
## statistic-key: downheap()
##   comparisions      swaps
##       34895918        17292967
#####
## statistic-key: upheap()
##   comparisions      swaps
##       1277645          1277645
#####

```

Abbildung 7: Diese Abbildung zeigt die Statistiken für das Sortieren von 10.000, 100.000, 1.000.000 Elementen

Übung 3

```
test_already_ascending_sorted()
#####
## statistic-context-provider
##
## statistics-contexts-count:1
##
#####
## statistic-context-key: heaparraylistimpl elements[1000]
## start-date: 23:50:19:403
## end-date: 23:50:19:409
#####
## statistic-key: downheap()
##      comparisions      swaps
##      14998             7317
#####
## statistic-key: upheap()
##      comparisions      swaps
##      0                 0
#####
```

Abbildung 8: Diese Abbildung zeigt die Statistiken für das Sortieren von 1.000 Elementen die bereits aufsteigend sortiert sind

Übung 3

```

test_sort_ascending()
#####
## statistic-context-provider
##
## statistics-contexts-count:5
##
#####
## statistic-context-key: heaparraylistimpl elements[10]
## start-date: 23:50:19:410
## end-date: 23:50:19:410
#####
## statistic-key: downheap()
##   comparisions      swaps
##       34           12
#####
## statistic-key: upheap()
##   comparisions      swaps
##       7             7
#####

#####
## statistic-context-key: heaparraylistimpl elements[1000]
## start-date: 23:50:19:411
## end-date: 23:50:19:417
#####
## statistic-key: downheap()
##   comparisions      swaps
##   15034            7353
#####
## statistic-key: upheap()
##   comparisions      swaps
##   1204              1204
#####

#####
## statistic-context-key: heaparraylistimpl elements[10000]
## start-date: 23:50:19:417
## end-date: 23:50:19:448
#####
## statistic-key: downheap()
##   comparisions      swaps
##  216546            106707
#####
## statistic-key: upheap()
##   comparisions      swaps
##  12851              12851
#####

```

Abbildung 9: Diese Abbildung zeigt die Statistiken für das Sortieren von 10, 100, 1.000 Elementen

Übung 3

```
#####
## statistic-context-key: heaparraylistimpl elements[100000]
## start-date: 23:50:19:448
## end-date: 23:50:19:550
#####
## statistic-key: downheap()
##   comparisions      swaps
##     2830562         1399809
#####
## statistic-key: upheap()
##   comparisions      swaps
##     127540         127540
#####

#####
## statistic-context-key: heaparraylistimpl elements[1000000]
## start-date: 23:50:19:550
## end-date: 23:50:20:718
#####
## statistic-key: downheap()
##   comparisions      swaps
##    34898126        17293758
#####
## statistic-key: upheap()
##   comparisions      swaps
##    1273257         1273257
#####
```

Abbildung 10: Diese Abbildung zeigt die Statistiken für das Sortieren von 10.000, 100.000, 1.000.000 Elementen

Übung 3

```

test_already_descending_sorted()
#####
## statistic-context-provider
##
## statistics-contexts-count:1
##|
#####
## statistic-context-key: heaparraylistimpl elements[1000]
## start-date: 23:50:20:720
## end-date: 23:50:20:721
#####
## statistic-key: downheap()
##      comparisions      swaps
##      14998              7317
#####
## statistic-key: upheap()
##      comparisions      swaps
##              0              0
#####

#####
## statistic-context-provider
##
## statistics-contexts-count:0
##

#####
## statistic-context-provider
##
## statistics-contexts-count:1
##
#####
## statistic-context-key: heaparraylistimpl elements[0]
## start-date: 23:50:20:722
## end-date: 23:50:20:722
#####

```

Abbildung 11: Diese Abbildung zeigt die Statistiken für das Sortieren 1.000 Elementen die bereits absteigend sortiert sind

Übung 3

2.4 QuickSorter

Folgend ist die Dokumentation für den QuickSorter angeführt.

Diese Implementierung soll ebenfalls das Interface `Sorter<E>` implementieren und die `CodeStatistics` verwenden.

Entweder soll der Algorithmus so gewählt werden dass er aufsteigend und absteigend sortieren kann, oder die Liste soll bei der inversen Sortierung mittels `Collections.reverse(list)` umgedreht werden.

Ansonsten ist der QuickSort Algorithmus wie bekannt zu implementieren.

2.4.1 Source Code

Folgend ist der Source der QuickSort Implementierung angeführt.

../src/main/java/at/fhooe/swe4/lab3/sort/quick/QuickSorter.java

```

1 package at.fhooe.swe4.lab3.sort.quick;
2
3 import java.util.Arrays;
4 import java.util.Collections;
5 import java.util.List;
6
7 import at.fhooe.swe4.lab3.sort.api.Sorter;
8 import at.fhooe.swe4.lab3.stat.CodeStatistics;
9 import at.fhooe.swe4.lab3.stat.DefaultStatisticsProviderImpl;
10 import at.fhooe.swe4.lab3.stat.api.StatisticsProvider;
11
12 /**
13  * This is the Sorter implementation for the quicksort algorithm
14  *
15  * @author Thomas Herzog
16  *
17  * @param <V>
18  *         the values type of the to sort elements
19  */
20 public class QuickSorter<V extends Comparable<V>> implements Sorter<V> {
21
22     private final StatisticsProvider statProvider = new DefaultStatisticsProviderImpl();
23
24     public QuickSorter() {
25     }
26
27     @SuppressWarnings("unchecked")
28     @Override
29     public V[] sort(final V[] array, final SortType sorterType) {
30         if (array == null) {
31             throw new IllegalArgumentException("Cannot sort null array");
32         }
33         final List<V> result = sort(Arrays.asList(array), sorterType);
34         return (V[]) result.toArray();
35     }
36
37     @Override
38     public List<V> sort(List<V> list, SortType sorterType) {
39         if (sorterType == null) {
40             throw new IllegalArgumentException("SorterType not defined");
41         }
42         if (list == null) {
43             throw new IllegalArgumentException("Cannot sort null list");
44         }
45         statProvider.initContext(new StringBuilder(this.getClass().getSimpleName()).append(" elements[").append(
46             list.size()).append("]").toString());
47         quicksort(list, 0, (list.size() - 1));
48         if (SortType.DESENDING.equals(sorterType)) {
49             Collections.reverse(list);
50         }
51         return list;
52     }
53
54     /**
55     * Performs a quicksort in ascending order.
56     *
57     * @param values
58     *         the values to be sorted
59     * @param start
60     *         the start index
61     * @param end
62     *         the end index
63     */
64     private void quicksort(final List<V> values, final int start, final int end) {
65         final CodeStatistics stat = statProvider.getCtx().byKey("quicksort", Boolean.TRUE);
66         int i = start;
67         int k = end;
68
69         if ((end - start) >= 1) {
70             V pivot = values.get(start);
71             while (k > i) {
72                 while ((values.get(i).compareTo(pivot) <= 0) && (i <= end) && (k > i)) {
73                     stat.incIf();
74                     i++;
75                 }

```

Übung 3

```

75         while ((values.get(k).compareTo(pivot) > 0) && (k >= start) && (k >= i)) {
76             stat.incIf();
77             k--;
78         }
79         if (k > i) {
80             stat.incSwap();
81             swap(values, i, k);
82         }
83     }
84     stat.incSwap();
85     swap(values, start, k);
86     quicksort(values, start, k - 1);
87     quicksort(values, k + 1, end);
88 }
89 }
90
91 /**
92  * Swaps the elements at the indexes
93  *
94  * @param values
95  *         the array list where to swap elements
96  * @param i
97  *         the first index
98  * @param j
99  *         the second index
100  */
101 private void swap(final List<V> values, final int i, final int j) {
102     final V tmp = values.get(i);
103     values.set(i, values.get(j));
104     values.set(j, tmp);
105 }
106
107 @Override
108 public StatisticsProvider getStatistitcs() {
109     return statProvider.endContext();
110 }
111 }

```

2.4.2 Tests

Folgend sind die Test für die QuickSort Implementierung angeführt. Diese Tests können einfach in einem eclipse erneut ausgeführt und reproduziert werden, obwohl anzumerken ist, dass die Zeiten abhängig von den zu Verfügung stehenden Ressourcen abhängig sind.