SWE 4x	**Übung zu Softwareentwicklung mit modernen Plattformen 4**	**SS 2015, Übung 3**

Abgabetermin: SA in der KW 17

☐ **Gr. 1, E. Pitzer** **Name** _____ **Aufwand in h** _____

☐ **Gr. 2, F. Gruber-Leitner**

Punkte _____ **Kurzzeichen Tutor / Übungsleiter** _____ **/** _____

Das Problem von Richard H. (9 Punkte)

Implementieren Sie einen effizienten Algorithmus in Java um die „5-glatten" Zahlen bis zu einer Schranke *n* zu finden. Das sind alle Zahlen, deren Primfaktoren kleiner gleich fünf sind. Anders gesagt, alle Zahlen, die sich als $2^x * 3^y * 5^z$darstellen lassen. Eine dritte Möglichkeit ist die Definition als sogenannte Hammingfolge *H*:

- $1 \in H$
- $h \in H \Rightarrow 2 \cdot h \in H \wedge 3 \cdot h \in H \wedge 5 \cdot h \in H$
- keine weiteren Zahlen sind Elemente von *H*

Die ersten 10 Hammingzahlen sind somit 1, 2, 3, 4, 5, 6, 8, 9, 10 und 12.

Die Implementierung sollte dabei effizient genug sein um z.B. die 10000-ste Hammingzahl (288325195312500000) in deutlich unter einer Sekunde zu berechnen.

Schlacht der Sortieralgorithmen (in Java) (6 + 6 + 3 Punkte)

Nachdem wir uns in der Übung wieder mit der Heap-Datenstruktur beschäftigt haben, kommen sicher Erinnerungen an die ersten beiden Semester wieder, wo wir uns mit Sortieralgorithmen beschäftigt haben. Insbesondere mit dem Heapsort- sowie dem Quicksort-Algorithmus. Implementieren Sie beide Algorithmen in Java auf einfache Integer Felder und vergleichen Sie sowohl die Anzahl der Elementvergleiche als auch die Anzahl der Vertauschungsoperationen.

a) Implementierung, Dokumentation und ausführliches Testen des HeapSort-Algorithmus auf Integer Felder.

b) Implementierung, Dokumentation und ausführliches Testen des QuickSort-Algorithmus auf Integer Felder.

c) Vergleichen Sie die beiden Implementierungen mit Hilfe von System.nanoTime() sowie durch Instrumentieren der Algorithmen um die Anzahl der Elementvergleiche und Vertauschungsoperationen (swaps) mit zu zählen. Erstellen Sie eine kleine Statistik für Felder bis zu einer Größe von mindestens 50000 Elementen z.B. alle Zweierpotenzen und führen Sie eine ausreichende Anzahl von Wiederholungen durch um eine statistisch Signifikante Aussage machen zu können.

1 Hammingfolge

1.1 Lösungsidee

Folgend ist die Lösungsidee für die Aufgabenstellung Berechnung einer Hammingfolge angeführt.
Da es sich hierbei lediglich um einen einzigen Algorithmus handelt soll dieser als Klassenmethode implementiert werden. Das diese Klasse lediglich diese Klasenmethode enthalten soll, soll in dieser Klasse ein Privater Konstruktor `private Hamming () {}` implementiert werden um zu verhindern, dass diese Klasse instanziert 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 NavigableSet<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 der Navigable<E> Instanz hinzu
7. Wiederhole Schritt 3 solange folgendes gilt: `resultList.size(i) < n`

Es soll 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 NavigableSet<E> liefert und es gleichzeitig aus dem NavigableSet<E> entfernt. Für das zu verwendende Interface NavigableSet<E> soll eine TreeSet<E> Instanz verwendet werden. Dadurch sollte der Container in seiner Größe beschränkt werden, was den Sortierungsaufwand des Containers minimal halten sollte. 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. Es ist nicht notwendig einen eigenen Comparator<E> zu implementieren da die natürliche Ordnung der BigInteger Instanzen ausreicht (Implementiert das Interface Comparable<E>).

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> calulcateHammingNumbers(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 hammming number and remove from set
49       final BigInteger currentValue = sortedSet.pollFirst();
50       // Add this element to result lsit
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_negativ() {
23     Hamming.calulcateHammingNumbers(-1);
24   }
25
26   @Test(expected = IllegalArgumentException.class)
27   public void test_invalid_count_zeor() {
28     Hamming.calulcateHammingNumbers(0);
29   }
30
31   @Test
32   public void test_caluclation() {
33     int count = 1;
34     final int factor = 10;
35     final int repeation = 6;
36     for (int i = 0; i < repeation; i++) {
37       count *= factor;
38       final long startMillis = System.currentTimeMillis();
39       final List<BigInteger> result = Hamming.calulcateHammingNumbers(count);
40       final long diffMillis = System.currentTimeMillis() - startMillis;
41       System.out.println("
    ------------------------------------------------------------------------");
42       System.out.println(new StringBuilder("Spend time in millis: '").append(diffMillis
    ).append("' for '")
43           .append(count).append("' hamming numbers").toString());
44       for (int j = 0; (j < result.size()) && (j < 20); j++) {
45         System.out.println(new StringBuilder().append(j + 1).append(": ").append(result
    .get(j)).toString());
46       }
47       System.out.println("
    ------------------------------------------------------------------------");
48       System.out.println("
    ------------------------------------------------------------------------");
49     }
50   }
51 }
```

1.3 Tests

Folgend sind die Tests der Aufgabenstellung Berechnen einer Hammingfolge angeführt.
Diese Test wurden mit Hilfe von JUnit in der Entwicklungsumgebung Eclipse implementiert, daher können diese Tests einfach in einer Eclipse Umgebung reproduziert werden.
Die angezeigten Berechnungszeiten sind abhängig von der verwendeten Hardware.

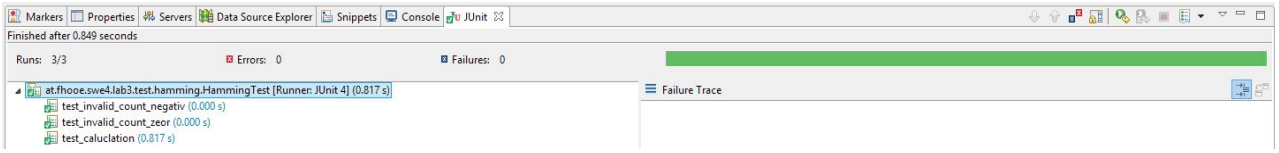

Abbildung 1: Diese Abbildung zeigt das Resultat der JUnit Tests im Eclipse

```
------------------------------------------------------------------
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, 100 Hammingzahlen

```
------------------------------------------------------------------
------------------------------------------------------------------
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, 10.000 Hammingzahlen

```
------------------------------------------------------------------
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, 1.000.000 Hammingzahlen

2 Sortieralgorithmen

2.1 Lösungsidee (Allgemein)

Folgend sind die Lösungsideen der Sortierlagorithmen 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 Compareable<E> implementieren, angewendet werden können.
Daher muss das Interface folgende Signatur vorweisen.public Sorter<E extends Comparable<E>> {...}

2.1.1 Source Code (Allgemein)

Folgend ist der Source des Interface Sorter 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 comparesTo method
42      * @param right
43      *            the parameter for lefts compareTomethod 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 getStatisitcs();
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 Statistiken zu ermitteln und auch einen Report dieser zu generieren.
Hierbei soll diese Code Statistik Ressourcen 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 Instanz, die Instanzen verwaltet, an deren `child.toString()` zu delegieren hat.

2.2.1 Source Code

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

```
}
```

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

```
package at.fhooe.swe4.lab3.stat;

import java.util.Calendar;
import java.util.Collections;
import java.util.Comparator;
import java.util.Set;
import java.util.SortedSet;
import java.util.TreeSet;

import org.apache.commons.collections4.CollectionUtils;
import org.apache.commons.collections4.Predicate;
import org.apache.commons.lang3.StringUtils;
import org.apache.commons.lang3.time.DateFormatUtils;

/**
 * This class represents a statistic context which is used for statistic code
 * analysis.
 *
 * @author Thomas Herzog
 *
 */
public class StatisticContext {
  private final String key;
  public Calendar startCalendar;
  public Calendar endCalendar;
  public final SortedSet<CodeStatistics> statisticsSet;

  /**
   * Default constructor which creates a context identified by the given
   * unique key.
   *
   * @param key
   *            the unique key for the created context
   * @throws IllegalArgumentException
   *             if the key is either null or an empty string
   */
  public StatisticContext(final String key) {
    super();
    if (StringUtils.isEmpty(key)) {
      throw new IllegalArgumentException("StatisticContext instance must be identified
    by an unique string key");
    }
    this.key = key;
    this.statisticsSet = new TreeSet<CodeStatistics>(new Comparator<CodeStatistics>() {
      @Override
      public int compare(CodeStatistics left, CodeStatistics right) {
        return left.getKey().compareTo(right.getKey());
      }
    });
  }

  /**
   * Gets the start calendar instance
   *
   * @return the start calendar
   */
  public Calendar getStartCalendar() {
    return startCalendar;
  }

```

```
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.
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 formatedStartDate() {
159      return formatDate(startCalendar);
160    }
161
162    /**
163     * @return the formatted end calendar string representation
164     * @see StatisticContext#formatDate(Calendar)
165     */
166    public String formatedEndDate() {
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   @Override
189   public int hashCode() {
190     final int prime = 31;
191     int result = 1;
192     result = prime * result + ((key == null) ? 0 : key.hashCode());
193     return result;
194   }
195
196   @Override
197   public boolean equals(Object obj) {
198     if (this == obj)
199       return true;
200     if (obj == null)
201       return false;
202     if (getClass() != obj.getClass())
203       return false;
204     StatisticContext other = (StatisticContext) obj;
205     if (key == null) {
206       if (other.key != null)
207         return false;
208     } else if (!key.equals(other.key))
209       return false;
210     return true;
211   }
212
213   /**
214    * Prints the statistic context and its code statistics
215    */
216   @Override
217   public String toString() {
218     final String ln = System.getProperty("line.separator");
219     final StringBuilder sb = new StringBuilder(500);
220     sb.append("
      ##############################################################################")
      .append(ln);
221     sb.append("## statistic-context-key: ").append(key).append(ln);
222     sb.append("## start-date: ").append(formatedStartDate()).append(ln);
223     sb.append("## end-date: ").append(formatedEndDate()).append(ln);
224     sb.append("
      ##############################################################################")
      .append(ln);
225     for (CodeStatistics statistics : statisticsSet) {
226       sb.append(statistics.toString()).append(ln);
227       sb.append("
      ##############################################################################")
      .append(ln);
228     }
229     return sb.toString();
230   }
231 }
```

../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
```

```
 *
 */
public class CodeStatistics {

  private int comparision;
  private int swaps;

  private final String key;

  /**
   *
   * @param key
   *            the key for this statistics data.
   * @throws IllegalArgumentException
   *             if the key is either null or an empty string
   */
  public CodeStatistics(final String key) {
    super();
    if (StringUtils.isEmpty(key)) {
      throw new IllegalArgumentException("Statistic instance must be identified by an
    unique string key");
    }
    this.key = key.trim().toLowerCase();
    clear();
  }

  /**
   * The key of this instance
   *
   * @return the instance key
   */
  public String getKey() {
    return key;
  }

  /**
   * Increase the comparison counter.
   *
   * @return the current instance
   */
  public CodeStatistics incIf() {
    comparision++;
    return this;
  }

  /**
   * Increases the swap counter
   *
   * @return the current instance
   */
  public CodeStatistics incSwap() {
    swaps++;
    return this;
  }

  /**
   * Clears the code statistic by setting all counters to '0'
   *
   * @return the current instance
   */
  public CodeStatistics clear() {
    comparision = 0;
    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   }
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.ArrayList;
4  import java.util.Calendar;
5  import java.util.HashMap;
6  import java.util.List;
7  import java.util.Map;
8  import java.util.Map.Entry;
9
10 import org.apache.commons.collections4.CollectionUtils;
```

```
11 import org.apache.commons.collections4.Predicate;
12 import org.apache.commons.lang3.StringUtils;
13
14 import at.fhooe.swe4.lab3.stat.api.StatisticsProvider;
15
16 /**
17  * This is the default statistics provider implementation.
18  *
19  * @author Thomas Herzog
20  *
21  */
22 public class DefaultStatisticsProviderImpl implements StatisticsProvider {
23
24   private StatisticContext ctx = null;
25   private final Map<String, StatisticsProvider> providers;
26   private final List<StatisticContext> statContextList = new ArrayList<StatisticContext
       >();
27
28   /**
29    * Default constructor which creates an context with the given key
30    *
31    * @param contextKey
32    *            the key for the initial context
33    */
34   public DefaultStatisticsProviderImpl() {
35     super();
36     this.providers = new HashMap<String, StatisticsProvider>();
37   }
38
39   @Override
40   public StatisticsProvider initContext(final String contextKey) {
41     if (ctx != null) {
42       endContext();
43     }
44     if (StringUtils.isEmpty(contextKey)) {
45       throw new IllegalArgumentException("Context key must be given fopr context");
46     }
47     ctx = new StatisticContext(contextKey.trim().toLowerCase());
48     ctx.setStartCalendar(Calendar.getInstance());
49     statContextList.add(ctx);
50     return this;
51   }
52
53   @Override
54   public StatisticsProvider endContext() {
55     if (ctx != null) {
56       ctx.setEndCalendar(Calendar.getInstance());
57       ctx = null;
58     }
59     return this;
60   }
61
62   @Override
63   public StatisticsProvider removeContext(final String ctxDelKey) {
64     final StatisticContext ctxDel = CollectionUtils.find(statContextList, new Predicate
       <StatisticContext>() {
65       @Override
66       public boolean evaluate(StatisticContext object) {
67         return object.getKey().equals(ctxDelKey.trim().toLowerCase());
68       }
69     });
70     if (ctxDel != null) {
71       statContextList.remove(ctxDel);
```

```
72      }
73      return this;
74    }
75
76    @Override
77    public StatisticsProvider takeOver(final String key, final StatisticsProvider
        provider) {
78      if (StringUtils.isEmpty(key) || (provider == null)) {
79        throw new IllegalArgumentException("Key and provider must be given");
80      }
81      this.providers.put(key, provider);
82      return this;
83    }
84
85    @Override
86    public StatisticContext getCtx() {
87      return ctx;
88    }
89
90    @Override
91    public String toString() {
92      final String ln = System.getProperty("line.separator");
93      final StringBuilder sb = new StringBuilder(500);
94      sb.append("
       ##########################################################################################")
       .append(ln);
95      sb.append("## statistic-context-provider").append(ln).append("##").append(ln);
96      sb.append("## statistics-contexts-count:").append(statContextList.size()).append(ln
       ).append("##").append(ln);
97      for (StatisticContext ctx : statContextList) {
98        sb.append(ctx.toString()).append(ln);
99      }
100     // Other providers
101     if (!providers.isEmpty()) {
102       sb.append("Managed providers:").append(ln);
103       for (Entry<String, StatisticsProvider> entry : providers.entrySet()) {
104         sb.append("Key:").append(entry.getKey()).append(ln);
105         sb.append(entry.getValue().toString()).append(ln);
106       }
107       sb.append("
       ##########################################################################################")
       .append(ln);
108     }
109
110     return sb.toString();
111   }
112 }
```

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 HeapSorter 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
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 getStatisitcs();
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
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
67           : originalArrayValues.length;
68       statProvider.initContext(new StringBuilder(this.getClass().getSimpleName()).append(
       " elements[").append(size)
69           .append("]").toString());
70       if (size > 0) {
71         container = new ArrayList<V>(size);
72         final CodeStatistics stat = statProvider.getCtx().newStatistic("init(array)");
73         for (V value : originalArrayValues) {
74           enqueue(value);
75         }
76       } else {
77         container = new ArrayList<V>(0);
78       }
79     }
80
81     @Override
82     public void init(final Collection<V> originalIterableValues, final HeapType heapType)
         {
83       this.heapType = heapType;
84       final int size = (originalIterableValues == null) ? 0 : originalIterableValues.size
       ();
85       statProvider.initContext(new StringBuilder(this.getClass().getSimpleName()).append(
       " elements[").append(size)
86           .append("]").toString());
87       if (size > 0) {
88         container = new ArrayList<V>(size);
89         final Iterator<V> it = originalIterableValues.iterator();
90         while (it.hasNext()) {
91           enqueue(it.next());
92         }
93       } else {
94         container = new ArrayList<V>(0);
95       }
96     }
97
98     @Override
99     public void enqueue(final V value) {
100      container.add(value);
101      upHeap(container);
102    }
103
104    @Override
105    public V dequeue() {
106      final V value = container.get(0);
107      container.set(0, container.get(container.size() - 1));
108      downHeap(container);
109      container.remove(container.size() - 1);
```

```
110     return value;
111   }
112
113   @Override
114   public boolean hasNext() {
115     return container.size() > 0;
116   }
117
118   @Override
119   public int size() {
120     return container.size();
121   }
122
123   @Override
124   public List<V> toList() {
125     return new ArrayList<V>(container);
126   }
127
128   @Override
129   @SuppressWarnings("unchecked")
130   public V[] toArray() {
131     return (V[]) container.toArray();
132   }
133
134   @Override
135   public StatisticsProvider getStatisitcs() {
136     return statProvider.endContext();
137   }
138
139   // Private heap methods
140   /**
141    * Performs an up heap on the given heap represented by the given list
142    *
143    * @param container
144    *            the list representing the heap
145    */
146   private void upHeap(final List<V> container) {
147     final CodeStatistics stat = statProvider.getCtx().byKey("upHeap()", Boolean.TRUE);
148
149     int i = container.size() - 1;
150     V tmp = container.get(i);
151     while ((i != 0) && (heapType.compare(container.get(parent(i)), tmp))) {
152       stat.incIf().incSwap();
153       container.set(i, container.get(parent(i)));
154       i = parent(i);
155     }
156     container.set(i, tmp);
157   }
158
159   /**
160    * Performs an down heap on the given heap represented by the given list
161    *
162    * @param container
163    *            the list representing the heap
164    */
165   private void downHeap(final List<V> container) {
166     final CodeStatistics stat = statProvider.getCtx().byKey("downHeap()", Boolean.TRUE)
     ;
167     int idx = 0;
168     int largeIdx;
169     V tmp = container.get(0);
170     while (idx < (container.size() / 2)) {
171       int leftIdx = left(idx);
```

```
172       int rightIdx = right(idx);
173       stat.incIf();
174       if ((rightIdx < container.size()) && (heapType.compare(container.get(leftIdx),
      container.get(rightIdx)))) {
175         largeIdx = rightIdx;
176       } else {
177         largeIdx = leftIdx;
178       }
179       stat.incIf();
180       if (!heapType.compare(tmp, container.get(largeIdx))) {
181         break;
182       }
183       stat.incSwap();
184       container.set(idx, container.get(largeIdx));
185       idx = largeIdx;
186     }
187     container.set(idx, tmp);
188   }
189
190   // Private helper
191   /**
192    * Gets the parent index of the element on index i
193    *
194    * @param i
195    *            the index to get its parent index
196    * @return the parent index
197    */
198   private static int parent(final int i) {
199     return (i - 1) / 2;
200   }
201
202   /**
203    * Gets the left neighbor index of the element on index i
204    *
205    * @param i
206    *            the index to get its left neighbor index
207    * @return the left neighbor index
208    */
209   private static int left(final int i) {
210     return (i * 2) + 1;
211   }
212
213   /**
214    * Gets the right neighbor index of the element on index i
215    *
216    * @param i
217    *            the index to get its right neighbor index
218    * @return the right neighbor index
219    */
220   private static int right(final int i) {
221     return (i * 2) + 2;
222   }
223
224   @Override
225   public String toString() {
226     final int new_line_count = 10;
227     final StringBuilder sb = new StringBuilder();
228     sb.append(this.getClass().getName()).append("[size=").append(container.size()).
      append("]\n");
229     sb.append("idx[0 - ").append(new_line_count).append("]: ");
230     for (int i = 0; i < container.size(); i++) {
231       sb.append(container.get(i));
232       if ((i + 1) < container.size()) {
```

```
233         sb.append(", ");
234       }
235       if ((i > 0) && (i % new_line_count == 0)) {
236         final int idxEnd = ((i + new_line_count) < container.size()) ? (i +
    new_line_count)
237             : (container.size() - 1);
238         sb.append(System.getProperty("line.separator"));
239         sb.append("idx[").append(i + 1).append(" - ").append(idxEnd).append("]: ");
240       }
241     }
242     return sb.toString();
243   }
244 }
```

../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 getStatisitcs() {
56      return heap.getStatisitcs();
57    }
58
59    /**
60     * Converts the sorter type to the corresponding heap type.
61     *
62     * @param sortType
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 sortType) {
70      switch (sortType) {
71      case ASCENDING:
72        return HeapType.MIN_HEAP;
73      case DESCENDING:
74        return HeapType.MAX_HEAP;
75      default:
76        throw new IllegalArgumentException("SortType cannot bemapped to corresponding
       HeapType !!!");
77      }
78    }
79  }
```

2.3.2 Tests

Folgend sind die Tests für die HeapSorter Implementierung angeführt.
Diese Test wurden mit Hilfe von JUnit in der Entwicklungsumgebung Eclipse implementiert, daher können diese Tests einfach in einer Eclipse Umgebung reproduziert werden.
Die angezeigten Berechnungszeiten sind abhängig von der verwendeten Hardware.
Die Zeitmessungen wurden in einem eigenen Punkt zusammengefasst, da hier beide Sortieralgorithmen Algorithmen gegeneinander verglichen werden.

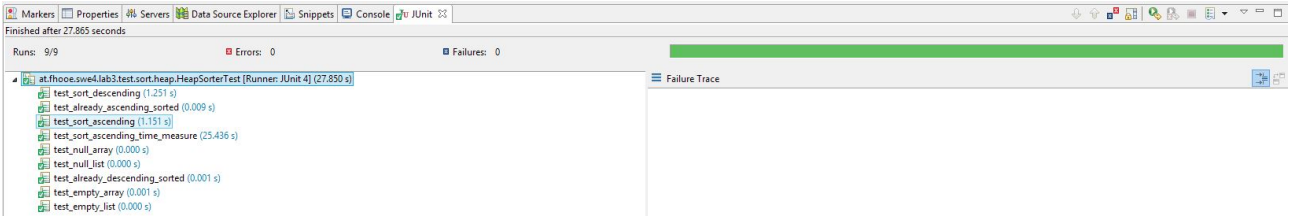

Abbildung 5: Diese Abbildung zeigt das Resultat der JUnit Tests im Eclipse

```
public void test_sort_descending()
################################################################################
## statistic-context-provider
##
## statistics-contexts-count:6
##
################################################################################
## statistic-context-key: heaparraylistimpl elements[10]
## start-date: 22:00:13:265
## end-date: 22:00:13:281
################################################################################
## statistic-key: downheap()
##    comparisions           swaps
##              34              12
################################################################################
## statistic-key: upheap()
##    comparisions           swaps
##               6               6
################################################################################

################################################################################
## statistic-context-key: heaparraylistimpl elements[100]
## start-date: 22:00:13:281
## end-date: 22:00:13:289
################################################################################
## statistic-key: downheap()
##    comparisions           swaps
##             866             411
################################################################################
## statistic-key: upheap()
##    comparisions           swaps
##             105             105
################################################################################

################################################################################
## statistic-context-key: heaparraylistimpl elements[1000]
## start-date: 22:00:13:289
## end-date: 22:00:13:297
################################################################################
## statistic-key: downheap()
##    comparisions           swaps
##           15058            7342
################################################################################
## statistic-key: upheap()
##    comparisions           swaps
##            1280            1280
################################################################################
```

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

```
################################################################################
## statistic-context-key: heaparraylistimpl elements[10000]
## start-date: 22:00:13:297
## end-date: 22:00:13:354
################################################################################
## statistic-key: downheap()
##    comparisions           swaps
##          216546          106733
################################################################################
## statistic-key: upheap()
##    comparisions           swaps
##           12748           12748
################################################################################

################################################################################
## statistic-context-key: heaparraylistimpl elements[100000]
## start-date: 22:00:13:354
## end-date: 22:00:13:495
################################################################################
## statistic-key: downheap()
##    comparisions           swaps
##         2830276         1399405
################################################################################
## statistic-key: upheap()
##    comparisions           swaps
##          127823          127823
################################################################################

################################################################################
## statistic-context-key: heaparraylistimpl elements[1000000]
## start-date: 22:00:13:495
## end-date: 22:00:14:453
################################################################################
## statistic-key: downheap()
##    comparisions           swaps
##        34895918        17292967
################################################################################
## statistic-key: upheap()
##    comparisions           swaps
##         1277645         1277645
################################################################################
```

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

```
public void test_already_ascending_sorted()
################################################################################
## statistic-context-provider
##
## statistics-contexts-count:1
##
################################################################################
## statistic-context-key: heaparraylistimpl elements[1000]
## start-date: 22:00:14:522
## end-date: 22:00:14:523
################################################################################
## statistic-key: downheap()
##    comparisions           swaps
##           14998            7317
################################################################################
## statistic-key: upheap()
##    comparisions           swaps
##               0               0
################################################################################
```

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

```
public void test_sort_ascending()
################################################################################
## statistic-context-provider
##
## statistics-contexts-count:6
##
################################################################################
## statistic-context-key: heaparraylistimpl elements[10]
## start-date: 22:00:14:523
## end-date: 22:00:14:531
################################################################################
## statistic-key: downheap()
##    comparisions           swaps
##              34              12
################################################################################
## statistic-key: upheap()
##    comparisions           swaps
##               7               7
################################################################################

################################################################################
## statistic-context-key: heaparraylistimpl elements[100]
## start-date: 22:00:14:531
## end-date: 22:00:14:531
################################################################################
## statistic-key: downheap()
##    comparisions           swaps
##             868             404
################################################################################
## statistic-key: upheap()
##    comparisions           swaps
##             118             118
################################################################################

################################################################################
## statistic-context-key: heaparraylistimpl elements[1000]
## start-date: 22:00:14:531
## end-date: 22:00:14:539
################################################################################
## statistic-key: downheap()
##    comparisions           swaps
##           15034            7353
################################################################################
## statistic-key: upheap()
##    comparisions           swaps
##            1204            1204
################################################################################
```

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

```
################################################################################
## statistic-context-key: heaparraylistimpl elements[10000]
## start-date: 22:00:14:539
## end-date: 22:00:14:571
################################################################################
## statistic-key: downheap()
##    comparisions           swaps
##          216546          106707
################################################################################
## statistic-key: upheap()
##    comparisions           swaps
##           12851           12851
################################################################################

################################################################################
## statistic-context-key: heaparraylistimpl elements[100000]
## start-date: 22:00:14:571
## end-date: 22:00:14:667
################################################################################
## statistic-key: downheap()
##    comparisions           swaps
##         2830562         1399809
################################################################################
## statistic-key: upheap()
##    comparisions           swaps
##          127540          127540
################################################################################

################################################################################
## statistic-context-key: heaparraylistimpl elements[1000000]
## start-date: 22:00:14:667
## end-date: 22:00:15:871
################################################################################
## statistic-key: downheap()
##    comparisions           swaps
##        34898126        17293758
################################################################################
## statistic-key: upheap()
##    comparisions           swaps
##         1273257         1273257
################################################################################
```

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

```
public void test_already_descending_sorted()
################################################################################
## statistic-context-provider
##
## statistics-contexts-count:1
##
################################################################################
## statistic-context-key: heaparraylistimpl elements[1000]
## start-date: 22:00:15:873
## end-date: 22:00:15:873
################################################################################
## statistic-key: downheap()
##    comparisions           swaps
##           14998            7317
################################################################################
## statistic-key: upheap()
##    comparisions           swaps
##               0               0
################################################################################
```

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

```
public void test_empty_array()
################################################################################
## statistic-context-provider
##
## statistics-contexts-count:0
##

public void test_empty_list()
################################################################################
## statistic-context-provider
##
## statistics-contexts-count:1
##
################################################################################
## statistic-context-key: heaparraylistimpl elements[0]
## start-date: 22:00:15:874
## end-date: 22:00:15:874
################################################################################
```

Abbildung 12: Diese Abbildung zeigt die Statistiken für das Sortieren einer leeren Liste und eines leeren Arrays

2.4 QuickSorter

Folgend ist die Lösungsidee für die QuickSorter Implementierung 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 QuickSorter 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[")
46         .append(list.size()).append("]").toString());
47     quicksort(list, 0, (list.size() - 1));
48     if (SortType.DESCENDING.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           }
76           while ((values.get(k).compareTo(pivot) > 0) && (k >= start) && (k >= i)) {
77             stat.incIf();
78             k--;
79           }
80           if (k > i) {
81             stat.incSwap();
82             swap(values, i, k);
83           }
84         }
85         stat.incSwap();
86         swap(values, start, k);
87         quicksort(values, start, k - 1);
88         quicksort(values, k + 1, end);
89       }
90     }
91
92     /**
93      * Swaps the elements at the indexes
94      *
95      * @param values
96      *            the array list where to swap elements
97      * @param i
98      *            the first index
99      * @param j
100     *            the second index
101     */
102    private void swap(final List<V> values, final int i, final int j) {
103      final V tmp = values.get(i);
104      values.set(i, values.get(j));
105      values.set(j, tmp);
106    }
107
108    @Override
109    public StatisticsProvider getStatisitcs() {
110      return statProvider.endContext();
111    }
112  }
```

2.4.2 Tests

Folgend sind die Test für die QuickSort Implementierung angeführt. Diese Test wurden mit Hilfe von JUnit in der Entwicklungsumgebung Eclipse implementiert, daher können diese Tests einfach in einer Eclipse Umgebung reproduziert werden.
Die angezeigten Berechnungszeiten sind abhängig von der verwendeten Hardware.
Die Zeitmessungen wurden in einem eigenen Punkt zusammengefasst, da hier beide Sortieralgorithmen Algorithmen gegeneinander verglichen werden.

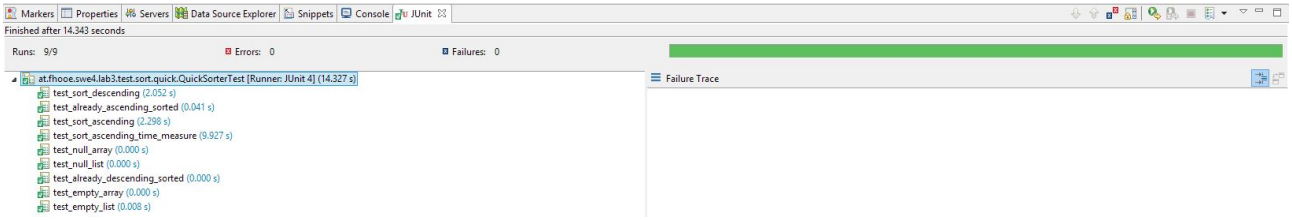

Abbildung 13: Diese Abbildung zeigt das Resultat der JUnit Tests im Eclipse

```
public void test_sort_descending()
################################################################################
## statistic-context-provider
##
## statistics-contexts-count:6
##
################################################################################
## statistic-context-key: quicksorter elements[10]
## start-date: 20:23:19:683
## end-date: 20:23:19:697
################################################################################
## statistic-key: quicksort
##    comparisions           swaps
##              29               9
################################################################################

################################################################################
## statistic-context-key: quicksorter elements[100]
## start-date: 20:23:19:697
## end-date: 20:23:19:698
################################################################################
## statistic-key: quicksort
##    comparisions           swaps
##             844             158
################################################################################

################################################################################
## statistic-context-key: quicksorter elements[1000]
## start-date: 20:23:19:698
## end-date: 20:23:19:704
################################################################################
## statistic-key: quicksort
##    comparisions           swaps
##           12317            2295
################################################################################
```

Abbildung 14: Diese Abbildung zeigt die Statistiken für das absteigende Sortieren von 10, 100, 1.000 Elementen

```
################################################################################
## statistic-context-key: quicksorter elements[10000]
## start-date: 20:23:19:704
## end-date: 20:23:19:734
################################################################################
## statistic-key: quicksort
##    comparisions           swaps
##          180212           31321
################################################################################

################################################################################
## statistic-context-key: quicksorter elements[100000]
## start-date: 20:23:19:734
## end-date: 20:23:19:964
################################################################################
## statistic-key: quicksort
##    comparisions           swaps
##         6327360          329070
################################################################################

################################################################################
## statistic-context-key: quicksorter elements[1000000]
## start-date: 20:23:19:964
## end-date: 20:23:21:699
################################################################################
## statistic-key: quicksort
##    comparisions           swaps
##       512740530         3333212
################################################################################
```

Abbildung 15: Diese Abbildung zeigt die Statistiken für das absteigende Sortieren von 10.000, 100.000, 1.000.000 Elementen

```
test_already_ascending_sorted()
################################################################################
## statistic-context-provider
##
## statistics-contexts-count:1
##
################################################################################
## statistic-context-key: quicksorter elements[1000]
## start-date: 20:48:35:896
## end-date: 20:48:35:945
################################################################################
## statistic-key: quicksort
##    comparisions           swaps
##          500499             999
################################################################################
```

Abbildung 16: Diese Abbildung zeigt die Statistiken für das aufsteigende Sortieren von 1.000 Elementen, die bereits aufsteigend sortiert sind

```
public void test_sort_ascending()
################################################################################
## statistic-context-provider
##
## statistics-contexts-count:6
##
################################################################################
## statistic-context-key: quicksorter elements[10]
## start-date: 20:42:08:277
## end-date: 20:42:08:278
################################################################################
## statistic-key: quicksort
##    comparisions           swaps
##              29               9
################################################################################

################################################################################
## statistic-context-key: quicksorter elements[100]
## start-date: 20:42:08:278
## end-date: 20:42:08:278
################################################################################
## statistic-key: quicksort
##    comparisions           swaps
##             844             158
################################################################################

################################################################################
## statistic-context-key: quicksorter elements[1000]
## start-date: 20:42:08:278
## end-date: 20:42:08:278
################################################################################
## statistic-key: quicksort
##    comparisions           swaps
##           12317            2295
################################################################################
```

Abbildung 17: Diese Abbildung zeigt die Statistiken für das aufsteigende Sortieren von 10, 100, 1.000 Elementen

```
################################################################################
## statistic-context-key: quicksorter elements[10000]
## start-date: 20:42:08:278
## end-date: 20:42:08:326
################################################################################
## statistic-key: quicksort
##    comparisions           swaps
##          180212           31321
################################################################################

################################################################################
## statistic-context-key: quicksorter elements[100000]
## start-date: 20:42:08:326
## end-date: 20:42:08:644
################################################################################
## statistic-key: quicksort
##    comparisions           swaps
##         6327360          329070
################################################################################

################################################################################
## statistic-context-key: quicksorter elements[1000000]
## start-date: 20:42:08:645
## end-date: 20:42:10:157
################################################################################
## statistic-key: quicksort
##    comparisions           swaps
##       512740530         3333212
################################################################################
```

Abbildung 18: Diese Abbildung zeigt die Statistiken für das aufsteigende Sortieren von 10.000, 100.000, 1.000.000 Elementen

```
test_already_descending_sorted()
################################################################################
## statistic-context-provider
##
## statistics-contexts-count:1
##
################################################################################
## statistic-context-key: quicksorter elements[1000]
## start-date: 20:48:38:753
## end-date: 20:48:38:754
################################################################################
## statistic-key: quicksort
##    comparisions           swaps
##          499999             999
################################################################################
```

Abbildung 19: Diese Abbildung zeigt die Statistiken für das absteigende Sortieren von 1.000 Elementen, die bereits absteigend sortiert sind

```
public void test_empty_array()
################################################################################
## statistic-context-provider
##
## statistics-contexts-count:1
##
################################################################################
## statistic-context-key: quicksorter elements[0]
## start-date: 20:42:10:159
## end-date: 20:42:10:159
################################################################################
## statistic-key: quicksort
##    comparisions           swaps
##               0               0
################################################################################


public void test_empty_list()
################################################################################
## statistic-context-provider
##
## statistics-contexts-count:1
##
################################################################################
## statistic-context-key: quicksorter elements[0]
## start-date: 20:42:10:160
## end-date: 20:42:10:160
################################################################################
## statistic-key: quicksort
##    comparisions           swaps
##               0               0
################################################################################
```

Abbildung 20: Diese Abbildung zeigt die Statistiken für das Sortieren von einer leeren Liste und eines leeren Array

3 Zeitauswertung

Folgend sind die Zeitmessungen für den HeapSorter angeführt, wobei diese mit 1.000.000 Zufallswerten im Bereich von 1 - 100.000 mit einem seed System.currentMillis() generiert wurden.
ACHTUNG: Diese Test sind sehr zeitintensiv und beanspruchen die Maschine stark. Abhängig von der Hardware könnten Probleme auftreten bzw. diese Tests sehr lange dauern.

```
public void test_sort_ascending_time_measure()
1: 1141 millis
2: 1001 millis
3: 1022 millis
4: 1041 millis
5: 996 millis
6: 993 millis
7: 1042 millis
8: 1174 millis
9: 1121 millis
10: 1016 millis
11: 1094 millis
12: 1019 millis
13: 990 millis
14: 1023 millis
15: 998 millis
16: 999 millis
17: 998 millis
18: 1059 millis
19: 996 millis
20: 1004 millis
--------------------
min:   990 millis
max:   1174 millis
range: 184 millis
--------------------
```

Abbildung 21: Diese Abbildung zeigt die Zeitmessungen des HeapSorter Algorithmus

```
public void test_sort_ascending_time_measure()
1: 548 millis
2: 543 millis
3: 334 millis
4: 322 millis
5: 313 millis
6: 314 millis
7: 323 millis
8: 352 millis
9: 319 millis
10: 326 millis
11: 318 millis
12: 309 millis
13: 318 millis
14: 328 millis
15: 326 millis
16: 328 millis
17: 311 millis
18: 321 millis
19: 320 millis
20: 329 millis
--------------------
min:   309 millis
max:   548 millis
range: 239 millis
--------------------
```

Abbildung 22: Diese Abbildung zeigt die Zeitmessungen des QuickSorter Algorithmus

Es ist zu sehen das der QuickSorter Algorithmus um den Faktor 3 schneller ist als der Heapsorter Algorithmus, daher ist dieser vorzuziehen. Bei mehreren Durchläufen hat sich auch gezeigt, dass die Zeiten der Sortierung der ersten Male beim QuickSorter länger dauern als die anderen Durchläufe, was ich auf die Laufzeitumgebung zurückführe.