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. Das diese Klasse lediglich diese Klasenmethode enthalten soll, soll in dieser Klasse ein Privater Konstruktor 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 ArrayList<E>welches die resultierenden Werte beinhaltet
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: $list.size(i) < (n + 4)$

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

1.2 Source-Code

Folgend ist der Implementierte Source und Test-Source angeführt.

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

```java
package at.fhooe.swe4.lab3.hamming;

import java.math.BigInteger;
import java.util.ArrayList;
import java.util.List;
import java.util.NavigableSet;
import java.util.TreeSet;

/**
 * Provides class methods which are used to handle hamming numbers.
 *
 * @author Thomas Herzog
 *
 */
public class Hamming {

  private static final BigInteger second = BigInteger.valueOf(2);
  private static final BigInteger three = BigInteger.valueOf(3);
  private static final BigInteger five = BigInteger.valueOf(5);

  /**
   * Not meant to be instantiated
   */
  private Hamming() {
    super();
  }

  /**
   * Calculates the 'count' hamming numbers.
   *
   * @param count
   *            the count of to calculate hamming numbers
   * @return the sorted list holding the hamming numbers
   */
  public static List<BigInteger> calulcateHammingNumbers(final int count) {
    // At least one is in the hamming list
    if (count <= 1) {
      throw new IllegalArgumentException("The count must be at least one !!!");
    }
    // Avoid grow of ArrayList
    final List<BigInteger> list = new ArrayList<BigInteger>(count);
    // Keeps calculated elements sorted
    final NavigableSet<BigInteger> sortedSet = new TreeSet<BigInteger>();
    // ONE is initial value
    sortedSet.add(BigInteger.ONE);
    // As long as we need to calculate
    while (list.size() != count) {
      // Get the next hammming number and remove from set
      final BigInteger currentValue = sortedSet.pollFirst();
      // Add this element to result lsit
      list.add(currentValue);
      // calculate next hamming numbers
      sortedSet.add(currentValue.multiply(second));
      sortedSet.add(currentValue.multiply(three));
      sortedSet.add(currentValue.multiply(five));
      System.out.println("size: " + sortedSet.size());
    }
    // Returned list is implicitly sorted
    return list;
  }
}
```

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

```java
package at.fhooe.swe4.lab3.test.hamming;

import java.math.BigInteger;
import java.util.List;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.JUnit4;

import at.fhooe.swe4.lab3.hamming.Hamming;

/**
 * This is the test for the calculating of the hamming numbers.
 *
 * @author Thomas Herzog
 *
 */
@RunWith(JUnit4.class)
public class HammingTest {

  @Test(expected = IllegalArgumentException.class)
  public void test_invalid_count_negativ() {
    Hamming.calulcateHammingNumbers(-1);
  }

  @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 '").
      append(count).append("' hamming numbers")
43            .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 Hammingfolge angeführt.
Aufgrund dessen das JUnit verwendet wurde und JUnit auch eine Report generiert wird hier auf das Einfügen der Tests verzichtet und nur der generierte JUnit Report verlinkt.

ACHTUNG: Da der Report mit einen relativen Pfad eingebunden wurde darf das Dokument nicht verschoben werden ohne das gewährleistet ist, dass das Verzeichnis "junit-report", welches die JUnit Reports enthält, wieder relativ gesehen an derselben Position ist

JUnit Report öffnen (index.html)

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 das Sortieren auf Integer Felder, jedoch sollen die Algorithmen so implementiert werden, dass sie auf Typen, die das Interface Compareable¡E¿ implementieren. Daher muss das Interface folgende Signatur vorweisen.

```
1 public Sorter<E extends Comparable<E>> {...}
```

2.1.1 Source Code

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

```java
package at.fhooe.swe4.lab3.sort.api;

import java.util.List;

import at.fhooe.swe4.lab3.sort.api.Heap.HeapType;
import at.fhooe.swe4.lab3.stat.api.StatisticsProvider;

/**
 * This interface specifies the sorter functionalities.
 *
 * @author Thomas Herzog
 *
 * @param <V>
 *            the values type of the collections or array elements
 */
public interface Sorter<V extends Comparable<V>> {
  /**
   * This enumeration specifies the sort order for a heap sort instance.
   *
   * @author Thomas Herzog
   *
   */
  public static enum SortType {
    /**
     * Will result in an ascending ordered result
     */
    DESCENDING,
    /**
     * Will result in an descending ordered result
     */
    ASCENDING;

    /**
     * Compares the two comparable instances.
     * <ul>
     * <li>
     * {@link SortType#DESCENDING} performs an x < 0 comparision</li>
     * <li>{@link SortType#ASCENDING} performs an x > 0 comparision</li>
     * </ul>
     *
     * @param left
     *            the instance which invokes the comparesTo method
     * @param right
     *            the parameter for lefts compareTomethod invocation
     * @return the proper result for the specified heap type
     */
    public <T extends Comparable<T>> boolean compare(T left, T right) {
      switch (this) {
      case DESCENDING:
        return left.compareTo(right) > 0;
      case ASCENDING:
        return left.compareTo(right) <= 0;
      default:
        throw new IllegalStateException("This enum is not handled here but should. enum=" + this.name());
      }
    }
  }

  /**
   * Sorts the given array.
   *
   * @param array
   *            the array to be sorted
   * @param sorterType
   *            the type of the sorting
   * @return the sorted array
   * @see SortType
   * @throws IllegalArgumentException
   *             if the array is null, or the {@link SortType} is null
   */
  public V[] sort(V[] array, SortType sorterType);

  /**
   * Sorts the given list5.
   *
   * @param list
   *            the list to be sorted
   * @param sorterType
   *            the type of the sorting
   * @return the sorted array
   * @see SortType
   * @throws IllegalArgumentException
   *             if the list is null, or the {@link SortType} is null
   */
  public List<V> sort(List<V> list, SortType sorterType);

  /**
   * Gets the statistics of the current instance
   *
   * @return the current statistics
   */
  public StatisticsProvider getStatisitcs();
}
```

2.2 Lösungsidee (Statistics)

Aufgrund dessen das 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 Statistics Provider enthalten soll.
   Die Implementierung soll es ermöglichen mehrere Statistic Kontexte zu verwalten.

2. **StatisticContext:** Die Klasse, welche einen Statstic Kontext darstellt soll.
   Dieser Kontext soll es ermöglichen mehrere CodeStatistic Instanzen pro Kontext zu verwalten.

3. **CodeStatistic:** Die Klasse, die die Code Statistic Informationen (swap, compare counts) halten soll

4. **DefaultStatisticProviderImpl:** Die default Implementierung des Interface StatisticProvider, welches die Funktionalitäten implementiert soll.

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

2.2.1 Source Code

Folgend ist der Source der Statistics 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   * 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();
59 }
```

../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
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
       key");
41      }
42      this.key = key;
43      this.statisticsSet = new TreeSet<CodeStatistics>(new Comparator<CodeStatistics>() {
44        @Override
45        public int compare(CodeStatistics left, CodeStatistics right) {
46          return left.getKey().compareTo(right.getKey());
47        }
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.
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    /**
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(formatedStartDate()).append(ln);
198      sb.append("## end-date: ").append(formatedEndDate()).append(ln);
199      sb.append("##########################################################################################").append(ln)
         ;
200      for (CodeStatistics statistics : statisticsSet) {
201        sb.append(statistics.toString()).append(ln);
202        sb.append("##########################################################################################").append(
         ln);
203      }
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 {
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   }
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<
       StatisticContext>() {
28     @Override
29     public int compare(StatisticContext left, StatisticContext right) {
30       return left.getStartCalendar().compareTo(right.getStartCalendar());
31     }
32   });
33
34   /**
35    * Default constructor which creates an context with the given key
36    *
37    * @param contextKey
38    *            the key for the initial context
39    */
40   public DefaultStatisticsProviderImpl() {
41     super();
42     this.providers = new HashMap<String, StatisticsProvider>();
43   }
44
45   @Override
46   public StatisticsProvider initContext(final String contextKey) {
47     if (ctx != null) {
48       endContext();
49     }
50     if (StringUtils.isEmpty(contextKey)) {
51       throw new IllegalArgumentException("Context key must be given fopr context");
52     }
53     ctx = new StatisticContext(contextKey.trim().toLowerCase());
54     ctx.setStartCalendar(Calendar.getInstance());
55     statContextSet.add(ctx);
56     return this;
57   }
58
59   @Override
60   public StatisticsProvider endContext() {
61     if (ctx != null) {
62       ctx.setEndCalendar(Calendar.getInstance());
63       ctx = null;
64     }
65     return this;
66   }
67
68   @Override
69   public StatisticsProvider removeContext(final String ctxDelKey) {
70     final StatisticContext ctxDel = CollectionUtils.find(statContextSet, new Predicate<StatisticContext>() {
71       @Override
72       public boolean evaluate(StatisticContext object) {
73         return object.getKey().equals(ctxDelKey.trim().toLowerCase());
74       }
75     });
76     if (ctxDel != null) {
77       statContextSet.remove(ctxDel);
78     }
79     return this;
80   }
81
82   @Override
83   public StatisticsProvider takeOver(final String key, final StatisticsProvider provider) {
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     sb.append("## statistic-context-ptovider").append(ln).append("##").append(ln);
102     sb.append("## statistics-contexts-count:").append(statContextSet.size()).append(ln).append("##").append(
        ln);
103     for (StatisticContext ctx : statContextSet) {
104       sb.append(ctx.toString()).append(ln);
105     }
106     // Other providers
107     if (!providers.isEmpty()) {
108       sb.append("Managed providers:").append(ln);
109       for (Entry<String, StatisticsProvider> entry : providers.entrySet()) {
110         sb.append("Key:").append(entry.getKey()).append(ln);
111         sb.append(entry.getValue().toString()).append(ln);
112       }
113       sb.append("################################################################################").append(
        ln);
114     }
115
116     return sb.toString();
117   }
118 }
```

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 sollen 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

```
1 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, was angibt ob aufsteigend oder absteigend sortiert.

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
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 : 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
126   @SuppressWarnings("unchecked")
127   public V[] toArray() {
128     return (V[]) container.toArray();
129   }
130
131   @Override
132   public StatisticsProvider getStatisitcs() {
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         largeIdx = rightIdx;
173       } else {
174         largeIdx = leftIdx;
175       }
176       stat.incIf();
177       if (!heapType.compare(tmp, container.get(largeIdx))) {
178         break;
179       }
180       stat.incSwap();
181       container.set(idx, container.get(largeIdx));
182       idx = largeIdx;
183     }
184     container.set(idx, tmp);
185   }
186
187   // Private helper
188   /**
189    * Gets the parent index of the element on index i
190    *
191    * @param i
192    *            the index to get its parent index
193    * @return the parent index
194    */
195   private static int parent(final int i) {
196     return (i - 1) / 2;
197   }
198
199   /**
200    * Gets the left neighbor index of the element on index i
201    *
202    * @param i
203    *            the index to get its left neighbor index
204    * @return the left neighbor index
205    */
206   private static int left(final int i) {
207     return (i * 2) + 1;
208   }
209
210   /**
211    * Gets the right neighbor index of the element on index i
212    *
213    * @param i
214    *            the index to get its right neighbor index
215    * @return the right neighbor index
216    */
217   private static int right(final int i) {
```

```
218     return (i * 2) + 2;
219   }
220
221   @Override
222   public String toString() {
223     final int new_line_count = 10;
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 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.MAX_HEAP;
73      case DESCENDING:
74        return HeapType.MIN_HEAP;
75      default:
76        throw new IllegalArgumentException("SortType cannot bemapped to corresponding HeapType !!!");
77      }
78    }
79  }
```

../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(
       list.size()).append("]").toString());
46     quicksort(list, 0, (list.size() - 1));
47     if (SortType.DESCENDING.equals(sorterType)) {
48       Collections.reverse(list);
49     }
50     return list;
51   }
52
53   /**
54    * Performs a quicksort in ascending order.
55    *
56    * @param values
57    *            the values to be sorted
58    * @param start
59    *            the start index
60    * @param end
61    *            the end index
62    */
63   private void quicksort(final List<V> values, final int start, final int end) {
64     final CodeStatistics stat = statProvider.getCtx().byKey("quicksort", Boolean.TRUE);
65     int i = start;
66     int k = end;
67
68     if ((end - start) >= 1) {
69       V pivot = values.get(start);
70       while (k > i) {
71         while ((values.get(i).compareTo(pivot) <= 0) && (i <= end) && (k > i)) {
72           stat.incIf();
73           i++;
74         }
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 getStatisitcs() {
109       return statProvider.endContext();
110     }
111 }
```

2.3.2 Tests

see junit index.html