1 Verschiebe Puzzle A*-Algorithmus

1.1 Lösungsidee

Folgend ist die Dokumentation für die Aufgabenstellung Verschiebe Puzzle mit A*-Algorithmus angeführt.
Es werden zwar Klassenspezifikationen angegeben, jedoch soll nicht stur diesen gefolgt werden, obwohl die Semantik beibehalten werden soll.
Bezüglich der konkreten Spezifikation der APIs und Implementierungen sei auf den Bereich *Source* verwiesen, wo einerseits beschrieben wird warum die API oder Implementierung so gewählt wurde und andererseits ist im angeführten Source die Javadoc enthalten. Ich finde es besser den Source über Javadoc zu kommentieren als in einem externen Dokument.

1.1.1 Maven Projekt

Dieses Mal soll das Projekt als Maven Projekt implementiert werden, da es sich hierbei um ein bekanntes und meist genutztes Framwork für Build Konfiguration handelt und es sehr einfach erlaubt Dependencies einzubinden und auch automatisch die Sources zur Verfügung stellt.
Dies sollte kein Problem darstellen, da Eclipse LUNA bereits mit Maven ausgelifert wird. Es soll gewährleistet werden, dass es keine speziellen Konfigurationen erfordert um das Projekt, in der von der FH zur Verfügung gestellten VM, zu importieren und zum Laufen zu bringen.
Jedoch sei angemerkt das es ein *Update Projekt* von Nöten sein wird um alle Dependencies von den Maven Repositories zu bekommen. Des Weiteren sollen keine Änderungen an der Standard *settings.xml* vorgenommen werden.

Für weitere Informationen bezüglich der Maven Konfiguration sei auf den Bereich *Sources* verwiesen, in dem die *pom.xml* angeführt ist. Dort ist auch beschrieben mit welchen Maven Build command das Projekt gebuildet werden soll, da es auch Konfigurationen geben soll, die einen report für die Tests erstellen.

1.1.2 JUnit Tests

Wie verlangt soll testgetrieben vorgegangen werden wobei anstatt einer riesigen Testklasse pro getesteter Methode eine eigene Test-Klasse implementiert werden, da es pro Testmethode sicherlich mehr als eine Test-Methode geben wird.
Bei den JUnit Tests soll sich an folgende Konvention gehalten werden:

1. **Test-Root-Package:** Alle Test-Klassen sollen im package *at.fh.ooe.swe.test.** zusammengefasst werden.
   Dadurch liegen alle Test-Klassen in einer eignen Domain und sind völlig unabhängig vom eigentlichen Source
2. **Test-API-Package:** Alle Test-Resourcen wie z.B.: abstrakte Test-Klassen sollen im Package *at.fh.ooe.swe.test.api.** liegen.
   Dadurch ist erkennbar das es sich hierbei um eine API für die Tests handelt.
3. **Test-Packages:** Alle tatsächlichen Tests sollen in Packages liegen die wie folgt aufgebaut sein sollen '*at.fh.ooe.swe.test.<sourcePackage>.<testedClassName> (at.fh.ooe.swe.test.puzzle.impl.boardImpl.*)*', wobei man sich an die Java Bean Convention zu halten hat.
4. **Test-Klassen:** Die Test-Klassen Name sollen wie folgt aufgebaut sein: '*<testedMethodName>Test (SolveTest, MakeMoveTest, usw.)*'
   Ein Spezialfall währen die Konstruktor Tests, die folgende Konvention folgen sollen. *<meaning>ConstructorTest (DefaultConstructorTest, CopyConstructorTest, usw.)*

5. **Test-Methoden:** Die Test-Methoden sollen so benannt werden, dass man leicht erkennen kann was getestet wird. Ein Präfix mit *test* ist zu vermeiden, da ja klar ist das es sich bei Methoden in Testklassen um Tests handelt. *(nullIterable, validAllFieldsRowPerRowLeftToRight, usw.)*

Für die JUnit Test soll JUnit4 verwendet werden wobei hierbei so viele Features wie möglich genutzt werden sollen. Da sicherlich bei den nächsten Übungen auch JUnit Test implementiert werden müssen, sollen alle Ressourcen, die bei anderen Tests wiederverwendet werden sollen in ein eigenes Projekt ausgelagert werden *junit-test-suite*, sodass diese Ressourcen einfach in anderen Projekten verwendet werden können. Es ist darauf zu achten, dass es absolut keine Abhängigkeiten zu dem eigentlichen Projekt gibt und das die Implementierungen so aufgebaut sind, dass sie für jedes anderes Projekt ohne Modifikation wiederverwendet werden können.

Da wir dazu gezwungen sind in dieser Dokumentation auch die Testresultate anzuführen und die Screenshots etwas zu wenig erscheinen, soll ein Logging implementiert werden, dass die Testresultate auf der Konsole ausgibt. Hierbei sollen *TestWatcher* implementiert werden, die einerseits die Invocation einer Testklasse sowie einer Testmethode auf die Konsole loggen. Hierbei soll ein bekanntes Logging Framework namens Log4j verwendet werden.

1.2 Source-Code

Folgend sind die Sources dieser Übung angeführt, sowie die Maven Konfiguration diese Projekts. Pro Source wird auch beschrieben warum die API oder Implementierung so gewählt wurde.

Für weitere Informationen bezüglich der Implementierungen sei auf die Javadoc verwiesen.

1.2.1 BoardListImpl.java

Folgend ist das Interface *Board.java* angeführt, welches die Spezifikation für die Board Implementierung darstellt. Es wurde beschlossen ein eigenes Interface zu spezifizieren, da es auch möglich sein soll den Container für die Verwaltung der Kacheln auszutauschen (`List<T>`, `T[]`, `usw.`)
Des weiteren wurde beschlossen eine Typdefinition zu verlangen, da es auch möglich sein soll andere Datentypen außer Integer für die Kacheln zu verwenden, den die einzige Voraussetzung ist, dass sich der Datentyp einer Rangordnung unterwerfen lässt, was über das Interface `Comparable<T>` gewährleistet wird. Es spielt hierbei keine Rolle um welchen Datentyp es sich handelt `Integer`, `BigInteger`, `Decimal`, `Character`, `usw.`, das einzig wichtige ist das der Datentyp das Interface `Comparable<T>` korrekt implementiert.

Listing 1: BoardListImpl.java

```
1  package at.fh.ooe.swe4.puzzle.api;
2
3  import java.util.NoSuchElementException;
4  import java.util.function.IntConsumer;
5  import java.util.stream.IntStream;
6
7  import at.fh.ooe.swe4.puzzle.exception.InvalidBoardIndexException;
8  import at.fh.ooe.swe4.puzzle.exception.InvalidMoveException;
9  import at.fh.ooe.swe4.puzzle.model.Position;
10
11 /**
12  *
13  * This is the interface specification for the puzzle board.<br>
14  * The indices on a board start with 1 and not with 0.<br>
15  * E.g.:<br>
16  * A board with the dimension 4X4 are build as follows.
17  * <ul>
18  * <li><b>upper left corner:</b> (1,1)</li>
19  * <li><b>lower right corner:</b> (4,4)</li>
20  * </ul>
21  *
22  *
23  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
24  * @date Apr 26, 2015
25  * @param <T>
26  *            the value type of the values on the board
27  */
28 public interface Board<T extends Comparable<T>> extends Comparable<Board<T>>, Cloneable {
29
30   /**
31    * Specifies the directions the empty tile can be moved.
32    *
33    * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
34    * @date Apr 26, 2015
35    */
36   public static enum Move {
37     UP, DOWN, LEFT, RIGHT;
38   }
39
```

```java
/**
 * Returns the tile at the position (rowIdx, colIdx).
 *
 * @param rowIdx
 *            the row index
 * @param colIdx
 *            the column index.
 * @return the found tile
 * @throws InvalidBoardIndexException
 *             if the indices are invalid
 */
public T getTile(int rowIdx, int colIdx);

/**
 * Sets an tile on the given position (rowIdx, colIdx).
 *
 * @param rowIdx
 *            the row index
 * @param colIdx
 *            the column index
 * @param value
 *            the value to be set on the defined position
 * @throws InvalidBoardIndexException
 *             if the indices are invalid
 */
public void setTile(int rowIdx, int colIdx, T value);

/**
 * Sets an empty tile on the given position (rowIdx, colIdx).
 *
 * @param rowIdx
 *            the row index
 * @param colIdx
 *            the column index
 * @throws InvalidBoardIndexException
 *             if the indices are invalid
 */
public void setEmptyTile(int rowIdx, int colIdx);

/**
 * Gets position of the empty tile (rowIdx, colIdx).<br>
 * If multiple empty tiles are present on this board then this method will
 * return the first occurrence of an empty tile.<br>
 * If no empty tile is present the returned position instance will contain
 * invalid indices.
 *
 * @return the position model with the indices, it will have the indices set
 *         to (-1, -1) if no empty tile could not be found
 */
public Position getEmptyTilePosition();

/**
 * Gets the Position of the tile with the given value.
 *
 * @param value
 *            the value to be searched on this board.
 * @return the found tile position
 * @throws NoSuchElementException
 *             if the value could not be found on the board
 */
public Position getTilePosition(final T value);

/**
```

```
103     * Gets the size of the board N where the board will have the dimensions N x
104     * N
105     *
106     * @return the size of this board
107     */
108    public int size();
109
110    /**
111     * Answers the question if this board is a valid board.<br>
112     * A board is invalid if one of the following conditions fit
113     * <ul>
114     * <li>no empty tile present</li>
115     * <li>tiles with duplicates values are present</li>
116     * </ul>
117     *
118     * @return true if this board is valid, false otherwise
119     */
120    public boolean isValid();
121
122    /**
123     * Shuffles the board tiles by performing random moves on the board.<br>
124     * It is ensured that the parity is kept (odd/even parity will be kept).
125     */
126    public void shuffle();
127
128    /**
129     * Moves the empty tile to the given position by switching the value on the
130     * given position with the empty tile position.<br>
131     * This method does not ensure that the move is a valid one, which means it
132     * is not ensured that the tile is only moved one position in any direction.
133     *
134     * @param rowIdx
135     *            the row index
136     * @param colIdx
137     *            the column index
138     * @throws InvalidBoardIndexException
139     *             if either the indices or the board is invalid
140     */
141    public void move(int rowIdx, int colIdx);
142
143    /**
144     * Moves the empty tile on step left.
145     *
146     * @throws InvalidMoveException
147     *             if the empty tile is already placed on the outer left column
148     */
149    public void moveLeft();
150
151    /**
152     * Moves the empty tile on step left.
153     *
154     * @throws InvalidMoveException
155     *             if the empty tile is already placed on the outer right column
156     */
157    public void moveRight();
158
159    /**
160     * Moves the empty tile on step right.
161     *
162     * @throws InvalidMoveException
163     *             if the empty tile is already placed on the top row
164     */
165    public void moveUp();
```

```
166
167   /**
168    * Moves the empty tile on step down.
169    *
170    * @throws InvalidMoveException
171    *             if the empty tile is already placed on the bottom row
172    */
173   public void moveDown();
174
175   /**
176    * Performs all of the moves of the empty tile defined by the given Iterable
177    * instance.
178    *
179    * @param moves
180    *            the Iterable instance holding the move positions for the empty
181    *            tile
182    * @throws InvalidMoveException
183    *             if the empty tile is tried to be moved out of the board
184    */
185   public void makeMoves(Iterable<Move> moves);
186
187   /**
188    * Calculates the parity of this board.<br>
189    * The parity is build as follows: <br>
190    * <sum_of_ordered_pairs> + <row_idx_empty_tile>
191    *
192    * @return the parity of this board
193    * @throws IllegalStateException
194    *             if this board is invalid
195    */
196   public int calculateParity();
197
198   // Force overwrite of clone
199   /**
200    * Performs an deep copy of the current instance.
201    *
202    * @return the copied instance
203    */
204   public Board<T> clone();
205 }
```

1.2.2 BoardImpl.java

Folgend ist die Implementierung für das Interface `Board.java` angeführt, welches als Container eine `ArrayList` verwendet. Dieser Container wurde gewählt da es sich im Hintergrund um ein `Array` handelt, daher indexierte Zugriffe möglich sind, dieser Container jedoch Methoden zur Verfügung stellt die den Zugriff bzw. die Handhabung dieses `Array` erleichtert.
Der Container wird mit der zu erwartenden Größe initialisiert um ein dynamisches wachsen, welches ein `Array.copy(...)` verursacht, zu vermeiden.
Dies ist zwar nicht zwingend notwendig aber ist als Best Practise anzusehen.

Listing 2: Board.java

```
1  package at.fh.ooe.swe4.puzzle.impl;
2
3  import java.util.ArrayList;
4  import java.util.Arrays;
5  import java.util.Collections;
6  import java.util.Comparator;
7  import java.util.List;
8  import java.util.NoSuchElementException;
9  import java.util.function.Consumer;
10 import java.util.function.Function;
11 import java.util.stream.Collectors;
12 import java.util.stream.IntStream;
13
14 import org.apache.commons.lang.StringUtils;
15
16 import at.fh.ooe.swe4.puzzle.api.Board;
17 import at.fh.ooe.swe4.puzzle.exception.InvalidBoardIndexException;
18 import at.fh.ooe.swe4.puzzle.exception.InvalidMoveException;
19 import at.fh.ooe.swe4.puzzle.model.Position;
20
21 /**
22  * This is the implementation of the {@link Board} interface.<br>
23  * This class provides possibility to define the tile value type which needs to
24  * implement {@link Comparable} interface.
25  *
26  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
27  * @date May 2, 2015
28  * @param <T>
29  *            the value type of the tile values
30  */
31 public class BoardListImpl<T extends Comparable<T>> implements Board<T> {
32
33   private final int size;
34   private final List<T> container;
35
36   /**
37    * The default constructor for this class which needs an size to be given.
38    *
39    * @param size
40    *            the size of the board, for the rows and the columns
41    * @throws IllegalArgumentException
42    *             if size <=0
43    */
44   public BoardListImpl(final int size) {
45     super();
46     if (size <= 0) {
47       throw new IllegalArgumentException("Size must be greater than 0");
48     }
49     this.size = size;
50     final int containerSize = (int) Math.pow(size, 2);
```

```
51      container = new ArrayList<T>(containerSize);
52      // Init with null values
53      for (int i = 0; i < containerSize; i++) {
54        container.add(null);
55      }
56    }
57
58    /**
59     * This constructor provides the size and container which contains the board
60     * values, where the container will be copied and cannot be modified from
61     * the outside.<br>
62     * The container must be a flat representation of the board.<br>
63     * It's size must be size*size which is the same as rows*columns.<br>
64     * E.g.: A value positioned at (3,2) will have the container index
65     * (3-1)*size + (2-1)
66     *
67     * @param size
68     *            the size of the board
69     * @param container
70     *            the container holding the board values
71     * @throws IllegalArgumentException
72     *             <ul>
73     *             <li>size <= 0</li>
74     *             <li>container is null</li>
75     *             <li>container.size() != (size*size)</li>
76     *             </ul>
77     */
78    public BoardListImpl(final int size, final List<T> container) {
79      super();
80      if (size <= 0) {
81        throw new IllegalArgumentException("Size must be greater than 0");
82      }
83      if (container == null) {
84        throw new IllegalArgumentException("Container must not be null");
85      }
86      if (((int) Math.pow(size, 2)) != container.size()) {
87        throw new IllegalArgumentException("Container size does not correspond to given board
          ↪  size");
88      }
89      this.size = size;
90      this.container = new ArrayList<T>(container);
91    }
92
93    @Override
94    public T getTile(int rowIdx, int colIdx) {
95      return container.get(calculateContainerIdx(rowIdx, colIdx));
96    }
97
98    @Override
99    public void setTile(int rowIdx, int colIdx, T value) {
100     container.set(calculateContainerIdx(rowIdx, colIdx), value);
101   }
102
103   @Override
104   public void setEmptyTile(int rowIdx, int colIdx) {
105     setTile(rowIdx, colIdx, (T) null);
106   }
107
108   @Override
109   public Position getEmptyTilePosition() {
110     final int[] indices = IntStream.range(0, container.size())
111                     .filter(i -> container.get(i) == null)
112                     .toArray();
```

```
113     if (indices.length != 0) {
114       final int rowIdx = ((indices[0] / size) + 1);
115       final int colIdx = (indices[0] - ((rowIdx - 1) * size) + 1);
116       return new Position(rowIdx, colIdx);
117     }
118     return new Position(-1, -1);
119   }
120
121   @Override
122   public Position getTilePosition(final T value) {
123     // row iteration
124     for (int i = 1; i <= size(); i++) {
125       // column iteration
126       for (int j = 1; j <= size(); j++) {
127         final T tile = getTile(i, j);
128         if (((tile == null) && (value == null)) || ((tile != null) && (tile.equals(value)))) {
129           return new Position(i, j);
130         }
131       }
132     }
133     throw new NoSuchElementException("Tile with value '" + ((value == null) ? "null" :
      ↪   value.toString()) + "' does not exist on the board");
134   }
135
136   @Override
137   public int size() {
138     return size;
139   }
140
141   @Override
142   public boolean isValid() {
143     return (container.stream()
144               .distinct()
145               .count() == (container.size())) && (container.stream()
146                                   .filter(element -> element == null)
147                                   .count() == 1);
148   }
149
150   @Override
151   public void shuffle() {
152     // We need to ensure the parity of the shuffled board
153     final boolean even = (calculateParity() % 2 == 0);
154     boolean shuffledEven = !even;
155
156     // fill list with the possible moves
157     final List<Move> moves = new ArrayList<Move>(Move.values().length * 4);
158     moves.addAll(Arrays.asList(Move.values()));
159     moves.addAll(Arrays.asList(Move.values()));
160     moves.addAll(Arrays.asList(Move.values()));
161     moves.addAll(Arrays.asList(Move.values()));
162
163     // as long as parity is not equal to original one
164     while (even != shuffledEven) {
165       // shuffle the possible moves
166       Collections.shuffle(moves);
167       moves.forEach(new Consumer<Move>() {
168         @Override
169         public void accept(Move direction) {
170           try {
171             // try to perform move
172             move(direction);
173           } catch (InvalidMoveException e) {
174             // expected because random movements could try to move
```

```
175             // empty tile out of the board
176           }
177         }
178       });
179       // recalculate the parity of the shuffled board
180       shuffledEven = (calculateParity() % 2 == 0);
181     }
182   }
183
184   @Override
185   public void move(int rowIdx, int colIdx) {
186     final Position position = getEmptyTilePosition();
187     setTile(position.rowIdx, position.colIdx, container.get(calculateContainerIdx(rowIdx,
      ↪  colIdx)));
188     setTile(rowIdx, colIdx, null);
189   }
190
191   @Override
192   public void moveLeft() {
193     move(Move.LEFT);
194   }
195
196   @Override
197   public void moveRight() {
198     move(Move.RIGHT);
199   }
200
201   @Override
202   public void moveUp() {
203     move(Move.UP);
204   }
205
206   @Override
207   public void moveDown() {
208     move(Move.DOWN);
209   }
210
211   @Override
212   public void makeMoves(Iterable<Move> moves) {
213     if (moves == null) {
214       throw new InvalidMoveException("Cannot perform moves because iterable instance is null");
215     }
216     moves.iterator()
217         .forEachRemaining(new Consumer<Move>() {
218           @Override
219           public void accept(Move t) {
220             move(t);
221           }
222         });
223   }
224
225   // Private helper
226   /**
227    * Check if the given indices are valid for this board.
228    *
229    * @param rowIdx
230    *            the row index
231    * @param colIdx
232    *            the column index
233    * @throws InvalidBoardIndexException
234    *             if at least one of the indices is invalid
235    */
236   private void checkForValidIndex(final int rowIdx, final int colIdx) {
```

```
237     if ((rowIdx > size) || (rowIdx <= 0) || (colIdx > size) || (colIdx <= 0)) {
238       throw new InvalidBoardIndexException("One of the indicies over or underlfows the border
          ↪  defined by size. rowIdx: " + rowIdx + " | colIdx: "
239           + colIdx);
240     }
241   }
242
243   /**
244    * Calculates the index for the backed container.
245    *
246    * @param rowIdx
247    *            the row index
248    * @param colIdx
249    *            the column index
250    * @return the calculated index
251    * @throws InvalidBoardIndexException
252    *             if at least one of the indexes is invalid
253    * @see BoardListImpl#checkForValidIndex(int, int)
254    */
255   private int calculateContainerIdx(final int rowIdx, final int colIdx) {
256     checkForValidIndex(rowIdx, colIdx);
257     return ((rowIdx - 1) * size) + (colIdx - 1);
258   }
259
260   /**
261    * Performs a move operation into the intended direction.<br>
262    * The move operation can be performed when the following conditions fit.
263    * <ul>
264    * <li>the board is valid</li>
265    * <li>the new position is valid</li>
266    * <li>the given {@link Move} enumeration is supported</li>
267    * </ul>
268    *
269    * @param direction
270    *            the direction to move to
271    * @throws InvalidMoveException
272    *             if the element cannot be moved to the intended direction
273    * @see BoardListImpl#calculateContainerIdx(int, int)
274    * @see BoardListImpl#getEmptyTilePosition()
275    */
276   private void move(final Move direction) {
277     if (direction == null) {
278       throw new InvalidMoveException("Cannot perform move operation with null direction");
279     }
280
281     int rowIdxDif = 0;
282     int colIdxDif = 0;
283     switch (direction) {
284     case UP:
285       rowIdxDif = -1;
286       break;
287     case DOWN:
288       rowIdxDif = 1;
289       break;
290     case LEFT:
291       colIdxDif = -1;
292       break;
293     case RIGHT:
294       colIdxDif = 1;
295       break;
296     default:
297       throw new IllegalArgumentException("Direction enum '" + direction.name() + "' is not
          ↪  handled");
```

```
298     }
299
300     try {
301       final Position oldPosition = getEmptyTilePosition();
302       checkForValidIndex(oldPosition.rowIdx, oldPosition.colIdx);
303       move((oldPosition.rowIdx + rowIdxDif), (oldPosition.colIdx + colIdxDif));
304     } catch (InvalidBoardIndexException e) {
305       throw new InvalidMoveException("Cannot move to the intended direction. direction: " +
          ↪  direction.name(), e);
306     }
307   }
308
309   /**
310    * Compares this instance to another by comparing the size of the board.
311    */
312   public int compareTo(Board<T> o) {
313     return Integer.valueOf(size)
314             .compareTo(o.size());
315   }
316
317   @Override
318   public int calculateParity() {
319     final Position position = getEmptyTilePosition();
320     final int emptyTileIdx = calculateContainerIdx(position.rowIdx, position.colIdx);
321     int parity = position.rowIdx;
322     for (int i = 0; i < container.size(); i++) {
323       // ignore empty tile
324       if (i != emptyTileIdx) {
325         for (int j = 0; j < i; j++) {
326           // ignore empty tile
327           if (j != emptyTileIdx) {
328             parity += (container.get(j)
329                       .compareTo(container.get(i)) > 0) ? 1 : 0;
330           }
331         }
332       }
333     }
334     return parity;
335   }
336
337   @Override
338   public Board<T> clone() {
339     return new BoardListImpl<T>(size, container);
340   }
341
342   @Override
343   public int hashCode() {
344     final int prime = 31;
345     int result = 1;
346     result = prime * result + ((container == null) ? 0 : container.hashCode());
347     return result;
348   }
349
350   @Override
351   public boolean equals(Object obj) {
352     if (this == obj)
353       return true;
354     if (obj == null)
355       return false;
356     if (getClass() != obj.getClass())
357       return false;
358     BoardListImpl<T> other = (BoardListImpl<T>) obj;
359     if (container == null) {
```

```
360       if (other.container != null)
361         return false;
362     } else if (!container.equals(other.container))
363       return false;
364     return true;
365   }
366
367   @Override
368   public String toString() {
369     final List<T> copy = new ArrayList<T>(container);
370     final int length = copy.stream()
371                 .max(Comparator.comparing(item -> ((item != null) ? item.toString()
372                                         .length() : 0)))
373                 .get()
374                 .toString()
375                 .length();
376     final StringBuilder sb = new StringBuilder(container.size() * length * 2);
377     sb.append(System.lineSeparator());
378     for (int i = 0; i < size(); i++) {
379       final List<T> result = copy.stream()
380                   .limit(size())
381                   .collect(Collectors.toList());
382       sb.append(result.stream()
383               .map(new Function<T, String>() {
384                 public String apply(T t) {
385                   final String value;
386                   int l = ((length > 1) && (length % 2 != 0)) ? (length + 1) : length;
387                   if (t != null) {
388                     value = t.toString();
389                   } else {
390                     value = " ";
391                   }
392                   return new StringBuilder("[").append(StringUtils.center(value, l))
393                                   .append("]")
394                                   .toString();
395                 };
396               })
397               .collect(Collectors.joining(" ")))
398         .append(System.lineSeparator());
399       copy.removeAll(result);
400     }
401     return sb.toString();
402   }
403 }
```

1.2.3 SlidingPuzzle.java

Folgend ist die Implementierung des SlidingPuzzle angeführt, welches die Boards versucht in die Zielkonfiguration zu überführen. Da es möglich sein soll nicht ur eine Zielkonfiguration zu handeln wurde die Implementierung erweitert um die Möglichkeit eine Zielkonfiguration zu übergeben.
Diese Klasse wurde nach dem Pattern *Fluent Interface* implementiert, da es dieser Pattern erlaubt hier eine gut verständliche Aufrufkette aufzubauen. Diese Aufrufkette liest sich wie Prosa wie in diesem Beispiel ersichtlich.

```
// Create new slide instance
SlidingPuzzle<Integer> slider = new SlidingPuzzle<>();
// start the slider with the initial board
slider.start(initialBoard)
  // Try to solve the board (returns SolutionHandler instance)
  .solve(goalBoard)
  // Starts the SolutionHandler instance
  .start()
  // Regsiter custom logger with custom level
  .registerLogger(myLog, Level.DEBUG)
  // Fill resulting moves in given list
  .fillMoves(moves)
  // prints the resulting moves via the logger
  .printMoves()
  // performs the moves on the inital board and logs inital/resulting board state
  .performMoves()
  // ends the SolutionHandler, returns related SlidingPuzzle instance
  .end()
  // Ends the SlidingPuzzle instance (reset state)
  .end();
```

Es wurde eine interne Klasse `SolutionHandler` implementiert, welche das Handling der Lösung übernimmt. Auch diese Implementierung verwendet *Fluent Interface*.
Um durchgehend mit demselben Typ zu arbeiten wird auch hier die Typdefinition verlangt, die der Typdefinition der Kacheln im `Board` entspricht. Damit ist gewährleistet, dass z.B.: eine `SlidingPuzzle<Integer>` Instanz nur auf `Board<Integer>` Instanzen angewendet werden kann und nicht z.B.: auf `Board<Character>` Instanzen.

Das macht es unmöglich einen Knoten mit einen bestimmten gesetzten Board zu erhalten, daher kann es auch keine Optimierung des Algorithmus geben, wobei bereits besuchte Knoten mit geringeren Kosten ausgenommen werden können.
Des Weiteren ist schwer aus einem `Set<T>` einen Knoten rauszubekommen. Man müsste über den gesamten Container iterieren und den Knoten suchen. Dieses Problem haben wir auch bei einer `PriorityQueue<T>`.

Daher wurden folgende Container gewählt:

1. **`PriorityQueue<T>`:** DIe Queue, die die Knoten mit den geringsten Kosten vorne anreiht und diese zuerst zurückliefert.

2. **`HashMap<Board<T>, SearchNode<T>>`:** Für die bereits untersuchten und offenen Knoten. Hierbei haben wir konstante Laufzeit beim Hinzufügen und Erhalten eines Knoten aus der Map, da hier über die Hash-Funktion `board.hashCode()` die Knoten in der Map adressiert werden. (Beinahe konstante Laufzeit aufgrund von Kollisionen)

Der Algorithmus versucht die Nachfolgeknoten des aktuell untersuchten Knoten in den Container offene wie auch geschlossenen Knoten zu finden und zu entscheiden ob diese Knoten der Container geringere Kosten besitzen. Wenn ja werden die aktuellen Nachfolgeknoten ignoriert und nicht weiter untersucht, da es schon einen besseren Weg zu diesem Board Status gibt.

Listing 3: BoardListImpl.java

```
1  package at.fh.ooe.swe4.puzzle.impl;
2
3  import java.util.ArrayList;
4  import java.util.HashMap;
5  import java.util.HashSet;
6  import java.util.Iterator;
7  import java.util.List;
8  import java.util.Map;
9  import java.util.PriorityQueue;
10 import java.util.Queue;
11 import java.util.Set;
12
13 import org.apache.commons.lang.StringUtils;
14 import org.apache.log4j.Level;
15 import org.apache.log4j.Logger;
16
17 import at.fh.ooe.swe4.puzzle.api.Board;
18 import at.fh.ooe.swe4.puzzle.api.Board.Move;
19 import at.fh.ooe.swe4.puzzle.exception.InvalidMoveException;
20 import at.fh.ooe.swe4.puzzle.exception.NoSolutionExcption;
21 import at.fh.ooe.swe4.puzzle.model.SearchNode;
22
23 /**
24  * This class is the solver for the game.
25  *
26  * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
27  * @date Apr 26, 2015
28  * @param <T>
29  *            The value type of the board tiles
30  */
31 public class SlidingPuzzle<T extends Comparable<T>> {
32
33   private Board<T> board;
34   private Board<T> goal;
35   private Queue<SearchNode<T>> queue;
36   private Map<Board<T>, SearchNode<T>> open;
37   private Map<Board<T>, SearchNode<T>> closed;
38
39   private boolean started = Boolean.FALSE;
40
41   /**
42    * This is the solution handler which handles the solution result
43    *
44    * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
45    * @date May 1, 2015
46    * @param <T>
47    *            The value type of the board tiles
48    */
49   public static final class SolutionHandler<T extends Comparable<T>> {
50     private final SearchNode<T> node;
51     private final SlidingPuzzle<T> slider;
52     private SearchNode<T> start;
53     private List<Move> moves;
54     private Logger log;
55     private Level level;
56
```

```
57      private boolean started = Boolean.FALSE;
58
59      /**
60       * Constructor which gets the current {@link SlidingPuzzle} instance and
61       * resulting {@link SearchNode} instance, where no solution has been
62       * found if the node is null.
63       *
64       * @param node
65       *            the {@link SearchNode} instance representing the result.
66       *            If null no result has been found.
67       * @param slider
68       *            the {@link SlidingPuzzle} instance which gets returned on
69       *            method call {@link SolutionHandler#end()}
70       */
71      public SolutionHandler(SearchNode<T> node, SlidingPuzzle<T> slider) {
72        super();
73        this.node = node;
74        this.slider = slider;
75      }
76
77      /**
78       * Starts the solution handling by creation the {@link Move} list.
79       *
80       * @return the current instance
81       * @throws IllegalStateException
82       *             if the solution handler is already started
83       * @throws NoSolutionExcption
84       *             if the set node is null
85       */
86      public SolutionHandler<T> start() throws NoSolutionExcption {
87        if (started) {
88          throw new IllegalStateException("The solution handler needs to be end before started
            ↪  again");
89        }
90        started = Boolean.TRUE;
91        if (node == null) {
92          throw new NoSolutionExcption("No solution found");
93        }
94        final Iterator<SearchNode<T>> it = node.iterator();
95        while (it.hasNext()) {
96          start = it.next();
97        }
98        moves = node.toMoves();
99        // init with default logging
100       this.log = Logger.getLogger(this.getClass());
101       this.level = Level.INFO;
102       return this;
103     }
104
105     /**
106      * Ability to provide a custom {@link Logger} instance and {@link Level}
107      *
108      * @param log
109      *            the {@link Logger} instance used for logging
110      * @param level
111      *            the {@link Level} instance defining the to use log level
112      * @return the current instance
113      * @throws IllegalStateException
114      *             if the solution handler hasn't been started yet
115      */
116     public SolutionHandler<T> registerLogger(final Logger log, final Level level) {
117       checkForStarted();
118       this.log = log;
```

```
119       this.level = level;
120       return this;
121     }
122
123     /**
124      * Prints the resulting moves via the {@link Logger}
125      *
126      * @return the current instance
127      * @throws IllegalStateException
128      *             if the solution handler hasn't been started yet
129      */
130     public SolutionHandler<T> printMoves() {
131       checkForStarted();
132       log.log(level, "Resulting moves:");
133       for (int i = 0; i < moves.size(); i++) {
134         log.log(level, new StringBuilder().append(String.format("%1$-" + ((moves.size() / 10) + 1)
          ↪  + "s", (i + 1)))
135                         .append(": ")
136                         .append(moves.get(i)
137                                 .name())
138                         .toString());
139       }
140       return this;
141     }
142
143     /**
144      * Performs the moves on the initial board which has been found via the
145      * given node.
146      *
147      * @return the current instance
148      * @throws IllegalStateException
149      *             if the solution handler hasn't been started yet
150      * @throws InvalidMoveException
151      *             if at least one of the resulting moves is invalid
152      * @see Board#makeMoves(Iterable)
153      */
154     public SolutionHandler<T> performMoves() {
155       checkForStarted();
156       final Board<T> board = start.getBoard()
157                       .clone();
158       start.getBoard()
159           .makeMoves(moves);
160       log.log(level, "Initial state:");
161       // .append(board.toString())
162       String[] boardStrings = null;
163       boardStrings = StringUtils.split(board.toString(), System.lineSeparator());
164       for (String string : boardStrings) {
165         log.log(level, string);
166       }
167       log.log(level, "Initial state after movements:");
168       boardStrings = StringUtils.split(start.getBoard()
169                       .toString(), System.lineSeparator());
170       for (String string : boardStrings) {
171         log.log(level, string);
172       }
173       return this;
174     }
175
176     @Deprecated
177     public List<Move> getMoves() {
178       return node.toMoves();
179     }
180
```

```
181     /**
182      * Fills the given list with the resulting moves.
183      *
184      * @param moves
185      *            the list to fill resulting moves in
186      * @return the current instance
187      * @throws IllegalArgumentException
188      *             if the given list if null
189      * @throws IllegalStateException
190      *             if the solution handler hasn't been started yet
191      */
192     public SolutionHandler<T> fillMoves(final List<Move> moves) {
193       checkForStarted();
194       if (moves == null) {
195         throw new IllegalArgumentException("Given moves list must not be null");
196       }
197       moves.addAll(this.moves);
198       return this;
199     }
200 
201     /**
202      * Ends the solution handling by reseting all members but the given node
203      * or slider.<br>
204      * This instance can be restarted by calling method
205      * {@link SolutionHandler#start}
206      *
207      * @return the related {@link SlidingPuzzle} instance
208      * @throws IllegalStateException
209      *             if the solution handler hasn't been started yet
210      */
211     public SlidingPuzzle<T> end() {
212       checkForStarted();
213       started = Boolean.FALSE;
214       this.level = null;
215       this.log = null;
216       this.moves = null;
217       return slider;
218     }
219 
220     /**
221      * Checks if the {@link SolutionHandler} instance has been started
222      *
223      * @throws IllegalStateException
224      *             if the instance hasn't been started but it is tried to
225      *             perform an action on it.
226      */
227     private void checkForStarted() {
228       if (!started) {
229         throw new IllegalStateException("The solution handler needs to be started before the
         ↪  solution can be handled");
230       }
231     }
232   }
233 
234   /**
235    * Default constructor which does not initializes this instance.<br>
236    * The {@link Board} instance to work on needs to be set via
237    * {@link SlidingPuzzleImpl#init(Board)}
238    *
239    * @see SlidingPuzzleImpl#init(Board)
240    */
241   public SlidingPuzzle() {
242     super();
```

```
243   }
244
245   /**
246    * Initializes the solver with the given board.<br>
247    * Keep in mind if the board is null or invalid the
248    *
249    * @param initial
250    *            the board to be resolved
251    * @return the current instance
252    * @throws IllegalStateException
253    *             if the slider has already been started
254    */
255
256   public SlidingPuzzle<T> start(final Board<T> initial) {
257     if (started) {
258       throw new IllegalStateException("The slider instance need to be end before restarted
        ↪  again.");
259     }
260     if ((initial == null) || (!initial.isValid())) {
261       throw new IllegalArgumentException("Cannot init this instance with an null or invalid
        ↪  board.");
262     }
263     this.started = Boolean.TRUE;
264     this.board = initial;
265     this.queue = new PriorityQueue<SearchNode<T>>();
266     this.open = new HashMap<Board<T>, SearchNode<T>>();
267     this.closed = new HashMap<Board<T>, SearchNode<T>>();
268     return this;
269   }
270
271   /**
272    * Resets this instance by setting all container to null.
273    *
274    * @throws IllegalStateException
275    *             if the solution handler hasn't been started yet
276    */
277   public SlidingPuzzle<T> end() {
278     checkForStarted();
279     this.started = Boolean.FALSE;
280     this.board = null;
281     this.goal = null;
282     this.queue = null;
283     this.closed = null;
284     return this;
285   }
286
287   /**
288    * Solves the set initial board and tries to calculate the moves to be made
289    * to transform the initial board to the goal board.
290    *
291    * @param goal
292    *            the goal board to resolve the initial board to
293    * @return the {@link SolutionHandler} instance which is responsible for
294    *         handling the result
295    * @throws NoSolutionExcption
296    *             <ul>
297    *             <li>goal is null</li>
298    *             <li>goal is invalid</li>
299    *             <li>goal has different size</li>
300    *             </ul>
301    */
302   public SolutionHandler<T> solve(final Board<T> goal) throws NoSolutionExcption {
303     checkForStarted();
```

```
304     // goal must represent valid board
305     if ((goal == null) || (!goal.isValid()) || ((goal.size() != board.size()))) {
306       throw new NoSolutionExcption("Cannot solve the board if the goal board is either null,
          ↪  invalid or of differen size");
307     }
308     this.goal = goal;
309
310     // validate parity of the two boards
311     if (!isResolvable()) {
312       return new SolutionHandler<T>(null, this);
313     }
314
315     // we found the solution right away
316     if (board.equals(goal)) {
317       return new SolutionHandler<T>(new SearchNode<>(0, null, board, goal, null), this);
318     }
319
320     SearchNode<T> current = new SearchNode<>(0, null, board, goal, null);
321     queue.add(current);
322     open.put(current.getBoard(), current);
323     // search as long nodes are left and solution hasn't been found
324     while (!queue.isEmpty()) {
325       // poll from queue
326       current = queue.poll();
327       // remove from open as well
328       open.remove(current.getBoard());
329       // get successors of current node by performing moves on it
330       final List<SearchNode<T>> successors = performMoves(current);
331       // handle found successors
332       for (SearchNode<T> successor : successors) {
333
334         // check if already on open
335         SearchNode<T> tmpNode = open.get(successor.getBoard());
336         // skip investigation if open node has lower costs
337         if ((tmpNode != null) && (tmpNode.compareTo(successor) <= 0)) {
338           continue;
339         }
340
341         // check if already on closed
342         tmpNode = closed.get(successor.getBoard());
343         // skip if closed has lower costs
344         if ((tmpNode != null) && (tmpNode.compareTo(successor) <= 0)) {
345           continue;
346         }
347
348         // check if we found the solution
349         if (successor.getBoard()
350                 .equals(goal)) {
351           return new SolutionHandler<T>(successor, this);
352         }
353
354         // add successors on open and queue
355         open.put(successor.getBoard(), successor);
356         queue.add(successor);
357       }
358
359       // remember investigated node
360       closed.put(current.getBoard(), current);
361     }
362     // no solution found should never occur
363     throw new IllegalStateException("Solution should have been found but wasn't. Maybe parity
        ↪  check failed");
364   }
```

```
365
366   // Private helper
367   /**
368    * Tries to perform all possible moves on the current node board
369    *
370    * @param parent
371    *            the parent which is the predecessor of the successor
372    * @return the list of found successors
373    */
374   private List<SearchNode<T>> performMoves(final SearchNode<T> parent) {
375     final List<SearchNode<T>> succesors = new ArrayList<SearchNode<T>>(Move.values().length);
376     for (Move direction : Move.values()) {
377       // Get board to move empty tile on
378       final Board<T> tmp = parent.getBoard()
379                       .clone();
380       try {
381         // perform the moves
382         switch (direction) {
383         case UP:
384           tmp.moveUp();
385           break;
386         case DOWN:
387           tmp.moveDown();
388           break;
389         case LEFT:
390           tmp.moveLeft();
391           break;
392         case RIGHT:
393           tmp.moveRight();
394           break;
395         default:
396           throw new UnsupportedOperationException("Direction with name '" + direction.name() + "'
              ↪  cannot not handled");
397         }
398         // Add found successor in case of valid move
399         succesors.add(new SearchNode<T>(parent.getCostsFormStart() + 1, parent, tmp, goal,
              ↪  direction));
400       } catch (InvalidMoveException e) {
401         // do nothing on invalid move
402       }
403     }
404
405     return succesors;
406   }
407
408   /**
409    * Answers the question if the set board is possible to be resolved to the
410    * set goal board.<br>
411    * It is if the parity is either even or odd on both boards.
412    *
413    * @return true if the set board is possible to be resolved to the set goal
414    *         board.
415    */
416   private boolean isResolvable() {
417     final int sourceParity = board.calculateParity() % 2;
418     final int targetParity = goal.calculateParity() % 2;
419     return (((sourceParity != 0) && (targetParity != 0)) || ((sourceParity == 0) && (targetParity
        ↪  == 0)));
420   }
421
422   /**
423    * Checks if the {@link SlidingPuzzle} instance has been started
424    *
```

```
425     * @throws IllegalStateException
426     *             if the instance hasn't been started but it is tried to
427     *             perform an action on it.
428     */
429    private void checkForStarted() {
430      if (!started) {
431        throw new IllegalStateException("The slider needs to be started before the solution can be
           ↪  handled");
432      }
433    }
434  }
```

1.2.4 Position.java

Folgend ist der Source des Models `Position` angeführt, welches dafür verwendet wird um die Position der leeren Kacheln zu liefern, da es mir zu wenig erschien nur mit der Spalte der leeren Kacheln zu arbeiten.

Listing 4: Position.java

```java
package at.fh.ooe.swe4.puzzle.model;

/**
 * Holds the position information of an tile on a board.
 *
 * @author Thomas Herzog <thomas.herzog@students.fh-hagenberg.at>
 * @date Apr 26, 2015
 */
public class Position {

  public final int rowIdx;
  public final int colIdx;

  public Position(int row, int column) {
    super();
    this.rowIdx = row;
    this.colIdx = column;
  }

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + colIdx;
    result = prime * result + rowIdx;
    return result;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Position other = (Position) obj;
    if (colIdx != other.colIdx)
      return false;
    if (rowIdx != other.rowIdx)
      return false;
    return true;
  }

  @Override
  public String toString() {
    return new StringBuilder("rowIdx: ").append(rowIdx)
                        .append(" | ")
                        .append("colIdx: ")
                        .append(colIdx)
                        .toString();
  }
}
```

1.2.5 SearchNode.java

Folgend ist der Source des Models `SearchNode<T>` angeführt, welches den Knoten für den A*-Algorithmus darstellt. Hierbei sei die Implementierung der Methoden `node.hashCode(); node.equals(other)` erwähnt, die nicht nur das referenzierte `Board<T>` verwendet sondern auch die Gesamtkosten.
Das Problem mit diesen Implementierungen ist, dass bei der Verwendung eines `Set<T>` Containers, welcher die geschlossenen Knoten enthält, Duplikate enthalten würde. Also Knoten mit demselben `Board` aber mit unterschiedlichen gesetzten Kosten.
Daher dürfen diese Knoten nicht in Containern verwendet werden, die deren `node.hashCode(); node.equals(Obj` verwenden.

1.3 Tests

2 Sortieralgorithmen

2.1 Lösungsidee (Allgemein)

2.1.1 Source Code (Allgemein)

2.2 Lösungsidee (Statistics)

2.2.1 Source Code

2.3 HeapSorter

2.3.1 Source Code

2.3.2 Tests

2.4 QuickSorter

2.4.1 Source Code

2.4.2 Tests

3 Zeitauswertung