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

Abgabetermin: SA in der KW 20

Gr. 1, E. Pitzer
Gr. 2, F. Gruber-Leitner

Name _____ **Aufwand in h** _____

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

Verschiebe-Puzzle – A*-Algorithmus (24 Punkte)

Ein sehr bekanntes und beliebtes Rätsel ist das Verschiebe-Puzzle, das oft auch als 8- bzw. 15-Puzzle bezeichnet wird. Das Spiel besteht aus 8 (15) Kacheln, die von 1 bis 8 (15) durchnummeriert sind, die auf einem 3x3-Spielfeld (4x4-Spielfeld) angeordnet sind. Da ein Feld frei bleibt, können gewisse Kacheln verschoben werden Die Aufgabe besteht nun darin, ausgehend von einer beliebigen Anordnung der Kacheln, diese ausschließlich durch Verschiebungen in die richtige Reihenfolge zu bringen (siehe nebenstehende Abbildung).

Eine Möglichkeit, dieses Problem zu lösen, ist Backtracking. Allerdings wird bei Anwendung dieses Verfahrens der Suchraum sehr groß, was zu nicht vertretbaren Rechenzeiten führt. Ein effizienter Algorithmus zur Lösung dieses Problems ist der sogenannte A*-Algorithmus, der von Peter Hart, Nils Nilsson und Bertram Raphael bereits 1968 entwickelt wurde. Eine übersichtliche Darstellung des Algorithmus findet man beispielsweise auf der deutschen Wikipedia unter http://de.wikipedia.org/wiki/A*-Algorithmus. Der A*-Algorithus wird oft zur Wegsuche bei Routenplanern eingesetzt. Er ist aber auch auf die hier angeführte Problemstellung anwendbar.

Die Grund A*-Algorithmus enumeriert grundsätzlich auch alle möglichen Lösungsvarianten, allerdings versucht er, zunächst den erfolgsversprechendsten Weg zum Ziel zu verfolgen. Erst dann werden weitere Varianten untersucht. Findet der Algorithmus auf diese Weise bereits frühzeitig eine Lösung, müssen viele Lösungsvarianten erst gar nicht evaluiert werden. Damit der Algorithmus beim Durchwandern des Lösungsraums in die erfolgsversprechendste Richtung weitergehen kann, benötigt er eine Abschätzung der Kosten, die auf dem verbleibenden Weg zum Ziel anfallen werden. In unserer Problemstellung kann für diese Kostenfunktion h(x) die Summe der Manhatten-Distanzen (= Distanz in x-Richtung + Distanz in y-Richtung) aller Kacheln zu ihrer Zielposition herangezogen werden. Wenn g(x) die Kosten von der Ausgangskonfiguration bis zur Konfiguration x bezeichnet, stellt f(x) = g(x) + h(x) eine Abschätzung der Kosten von der Ausgangs- zur Zielkonfiguration dar, wobei der Weg zum Ziel über x verläuft.

Implementieren Sie die Lösung in folgenden Schritten:

a) Gehen Sie bei der Implementierung testgetrieben vor. Implementieren Sie die nachfolgend angeführten Klassen Methode für Methode und geben Sie für jede Methode zumindest einen einfachen Testfall an. Erstellen Sie zunächst nur den Methodenrumpf mit einer Standardimplementierung, die nur syntaktisch korrekt sein muss. Implementieren Sie dann für diese Methode die Unittests, deren Ausführung zunächst fehlschlagen wird. Erweitern Sie anschließend die Implementierung der Methode so lange, bis alle Unittests durchlaufen. Erst wenn die Methoden-bezogenen Tests funktionieren, können Sie komplexere Tests erstellen.

   Eine Testsuite mit einigen Tests wird Ihnen auf der E-Learning-Plattform zur Verfügung gestellt. Erweitern Sie diese Testsuite so wie beschrieben. Ihre Implementierung muss die vorgegebenen und die von Ihnen hinzugefügten bestehen.

b) Implementieren Sie zunächst eine Klasse `Board`, die eine Board-Konfiguration repräsentieren kann und alle notwendigen Operationen auf einem Spielbrett unterstützt. `Board` soll folgende Schnittstelle aufweisen:

```
public class Board implements Comparable<Board> {
  // Board mit Zielkonfiguration initialisieren.
  public Board(int size);

  // Überprüfen, ob dieses Board und das Board other dieselbe Konfiguration aufweisen.
  public boolean equals(Object other);

  // <1, wenn dieses Board kleiner als other ist.
  // 0, wenn beide Boards gleich sind
  // >1, wenn dieses Board größer als other ist.
  public int compareTo(Board other);

  // Gibt die Nummer der Kachel an der Stelle (i,j) zurück, Indizes beginnen bei 1.
  // (1,1) ist somit die linke obere Ecke.
  // Wirft die Laufzeitausnahme InvalidBoardIndexException.
  public int getTile(int i, int j);

  // Setzt die Kachelnummer an der Stelle (i,j) zurück. Wirft die Laufzeitausnahmen
  //InvalidBoardIndexException und InvalidTileNumberException
  public void setTile(int i, int j, int number);

  // Setzt die Position der leeren Kachel auf (i,j)
  // Entsprechende Kachel wird auf 0 gesetzt.
  // Wirft InvalidBoardIndexException.
  public void setEmptyTile(int i, int j);

  // Zeilenindex der leeren Kachel
  public int getEmptyTileRow();

  // Gibt Spaltenindex der leeren Kachel zurück.
  public int getEmptyTileColumn();

  // Gibt Anzahl der Zeilen (= Anzahl der Spalten) des Boards zurück.
  public int size();

  // Überprüft, ob Position der Kacheln konsistent ist.
  public boolean isValid();

  // Macht eine tiefe Kopie des Boards.
  // Vorsicht: Referenztypen müssen neu allokiert und anschließend deren Inhalt kopiert werden.
  public Board copy();

  // Erzeugt eine zufällige lösbare Konfiguration des Boards, indem auf die bestehende
  // Konfiguration eine Reihe zufälliger Verschiebeoperationen angewandt wird.
  public void shuffle();

  // Verschiebt leere Kachel auf neue Position (row, col).
  // throws IllegalMoveException
  public void move(int row, int col);

  // Verschiebt leere Kachel nach links. Wirft Laufzeitausnahme IllegalMoveException.
  public void moveLeft();

  // Verschiebt leere Kachel nach rechts. Wirft IllegalMoveException.
  public void moveRight();

  // Verschiebt leere Kachel nach oben. Wirft IllegalMoveException.
  public void moveUp();
```

```
    // Verschiebt leere Kachel nach unten. Wirft IllegalMoveException.
    public void moveDown();

    // Führt eine Sequenz an Verschiebeoperationen durch. Wirft IllegalMoveException.
    public void makeMoves(List<Move> moves);
}
```

c) Zur Implementierung des A*-Algorithmus benötigt sie die Hilfsklasse `SearchNode`. Damit kann man den Weg von einem `SearchNode` zum Startknoten zurückverfolgen, da dieser mit seinem Vorgängerkonten verkettet ist. Ein `SearchNode` kennt die Kosten vom Startknoten bis zu ihm selbst. Ein `SearchNode` kann auch eine Schätzung für den Weg zum Zielknoten berechnen.

```
public class SearchNode implements Comparable<SearchNode> {

    // Suchknoten mit Board-Konfiguration initialisieren.
    public SearchNode(Board board);

    // Gibt Board-Konfiguration dieses Knotens zurück.
    public Board getBoard();

    // Gibt Referenz auf Vorgängerknoten zurück.
    public SearchNode getPredecessor();

    // Setzt den Verweis auf den Vorgängerknoten.
    public void setPredecessor(SearchNode predecessor);

    // Gibt Kosten (= Anzahl der Züge) vom Startknoten bis zu diesem Knoten zurück.
    public int costsFromStart():

    // Gibt geschätzte Kosten bis zum Zielknoten zurück.  Die Abschätzung
    // kann mit der Summe der Manhatten-Distanzen aller Kacheln erfolgen.
    public int estimatedCostsToTarget();

    // Setzt die Kosten vom Startknoten bis zu diesem Knoten.
    public void setCostsFromStart(int costsFromStart);

    // Gibt Schätzung der Wegkosten vom Startknoten über diesen Knoten bis zum Zielknoten zurück.
    public int estimatedTotalCosts();

    // Gibt zurück, ob dieser Knoten und der Knoten other dieselbe Board-Konfiguration darstellen.
    // Vorsicht: Knotenkonfiguration vergleichen, nicht die Referenzen.
    public boolean equals(Object other);

    // Vergleicht zwei Knoten auf Basis der geschätzten Gesamtkosten.
    // <1: Kosten dieses Knotens sind geringer als Kosten von other.
    //  0: Kosten dieses Knotens und other sind gleich.
    // >1: Kosten dieses Knotens sind höher als Kosten von other.
    public int compareTo(SearchNode other);

    // Konvertiert die Knotenliste, die bei diesem Knoten ihren Ausgang hat, in eine Liste von Zügen.
    // Da der Weg in umgekehrter Reihenfolge gespeichert ist, muss die Zugliste invertiert werden.
    public List<Move> toMoves();
}
```

d) Implementieren Sie schließlich den A*-Algorithmus in der Klasse `SlidingPuzzle`.

```
public class SlidingPuzzle {

   // Berechnet die Zugfolge, welche die gegebene Board-Konfiguration in die Ziel-Konfiguration
   // überführt. Wirft NoSolutionException (Checked Exception), falls es eine keine derartige
   // Zugfolge gibt.
   public List<Move> solve(Board board);

   // Gibt die Folge von Board-Konfigurationen auf der Konsole aus, die sich durch
   // Anwenden der Zugfolge moves auf die Ausgangskonfiguration board ergibt.
   public void printMoves(Board board, List<Move> moves);
}
```

Verwenden Sie bei Ihrer Lösung so weit wie möglich die Behälterklassen des JDK. Setzen Sie insbesondere bei der Implementierung des A*-Algorithmus (`SlidingPuzzle.solve()`) eine Prioritätswarteschlange (`PriorityQueue`) für die Speicherung Liste der offenen Knoten und eine sortierte Menge (`Set`) für die Verwaltung der Liste der geschlossenen Knoten ein.

1 Verschiebe-Puzzle – A*-Algorithmus

1.1 Lösungsidee

Als ersten Schritt der Lösung werden die, in der Angabe erwähnten, Klassen implementiert und die Testfälle erweitert. Dabei ist die Implementierung der meisten Methoden trivial.
Im folgenden werden nur mehr jede Lösungsideen angeführt, die mehr Überlegungen erfordern.

1.1.1 Board

Die Klasse *Board* verwendet als Datenspeicher eine *ArrayList*, wobei hier der Zeilen und Spaltenindex auf den Index in der *ArrayList* abgebildet werden muss.

1.1.2 SearchNode - calcManhattanDistance

Die Manhattan Distance wird in der Klasse *SearchNode* berechnet und wird für den A* Algorithmus benötigt.
Dabei wird das gesamte Board durchlaufen und für jedes Element die Abweichung zur Zielkonfiguration berechnet.
Die Zielposition kann folgendermaßen berechnet werden:

- Zeile: Nummer / Größe des Boards
- Spalte: Nummer % Größe des Boards

Die Manhattan Distance errechnet sich dann aus den Summen der Abweichungen der einzelnen Positionen.

1.1.3 SearchNode - toMoves

Um die Liste der Züge vom Start bis zur aktuellen Konfiguration zu berechnen, muss die verkette Liste aufgelöst werden.
Das Ergebnis muss dann noch umgedreht werden, da wir ja die Züge vom Start bis zum Ziel benötigen.

1.1.4 SlidingPuzzle - solve

Um das Verschiebe-Puzzle zu lösen wird, wie in der Angabe erwähnt, der A* Algorithmus angewendet.
Dieser benötigt eine **openQueue** und ein **closedSet**.

- openQueue: enthält die noch zu prüfenden Pfade sortiert nach ihren Kosten zum Ziel. (Darum kann hier eine *PriorityQueue* verwendet werden.)
- closedSet: enthält die bereits geprüften Pfade.

Ablauf:

- Zu Begin wird die Startkonfiguration in die **openQueue** eingefügt.
- In einer Schleife wird dann das oberste Element der Queue herausgenommen.
- Ist das Element die Zielkonfiguration, wurde eine Lösung gefunden.

- Wenn nicht, wird das Element in das **closedSet** eingefügt.
- Dann werden alle Nachfolger berechnet. (gültige Verschiebeoperationen anwenden)
- Jeder Nachfolger wird dann in die **openQueue** eingefügt, wenn er noch nicht betrachtet wurde oder seine Kosten geringer sind.

1.2 Sourcecode

Board.java

```
1  package at.lumetsnet.puzzle;
2
3  import java.util.ArrayList;
4  import java.util.List;
5  import java.util.Random;
6
7  import javax.management.RuntimeErrorException;
8
9  import at.lumetsnet.puzzle.exceptions.IllegalMoveException;
10 import at.lumetsnet.puzzle.exceptions.InvalidBoardIndexException;
11 import at.lumetsnet.puzzle.exceptions.InvalidTileNumberException;
12
13 public class Board implements Comparable<Board> {
14
15   /***
16    * Number of shuffle operations
17    */
18   private static final int SHUFFLE_COUNT = 100;
19
20   private final int size;
21   private final List<Integer> container;
22
23   public Board(int size) {
24     if (size <= 0) {
25       throw new IllegalArgumentException(
26           "Size has to be greater than zero.");
27     }
28     this.size = size;
29     container = new ArrayList<Integer>(size * size);
30     for (int i = 0; i < size * size - 1; i++) {
31       container.add(i + 1);
32     }
33     container.add(0);
34   }
35
36   /***
37    * Gets the tile index inside the container
38    *
39    * @param rowIdx
40    * @param colIdx
41    * @return
42    */
43   private int getTileIndex(int rowIdx, int colIdx) {
44     return ((rowIdx - 1) * size) + (colIdx - 1);
45   }
```

```
46
47  /***
48   * Checks for valid board indices
49   *
50   * @param rowIdx
51   * @param colIdx
52   * @throws InvalidBoardIndexException
53   */
54  private void checkBoardIndex(int rowIdx, int colIdx)
55      throws InvalidBoardIndexException {
56    if (rowIdx < 1 || rowIdx > size || colIdx < 1 || colIdx > size) {
57      throw new InvalidBoardIndexException(rowIdx + ", " + colIdx
58          + " are invalid indices");
59    }
60  }
61
62  /***
63   * gets the tile on the given position
64   *
65   * @param rowIdx
66   * @param colIdx
67   * @throws InvalidBoardIndexException
68   * @return
69   */
70  public int getTile(int rowIdx, int colIdx)
71      throws InvalidBoardIndexException {
72    checkBoardIndex(rowIdx, colIdx);
73    return container.get(getTileIndex(rowIdx, colIdx));
74  }
75
76  /***
77   * Sets the tile on the given index
78   *
79   * @param rowIdx
80   * @param colIdx
81   * @param number
82   * @throws InvalidBoardIndexException
83   * @throws InvalidTileNumberException
84   */
85  public void setTile(int rowIdx, int colIdx, int number)
86      throws InvalidBoardIndexException, InvalidTileNumberException {
87    checkBoardIndex(rowIdx, colIdx);
88    if (number < 0 || number > size * size) {
89      throw new InvalidTileNumberException("Tile number " + number
90          + " is not a valid tile number");
91    }
92    container.set(getTileIndex(rowIdx, colIdx), number);
93  }
94
```

```
95  /***
96   * Sets the empty tile on the given position
97   *
98   * @param rowIdx
99   * @param colIdx
100  * @throws InvalidBoardIndexException
101  */
102 public void setEmptyTile(int rowIdx, int colIdx)
103     throws InvalidBoardIndexException {
104   checkBoardIndex(rowIdx, colIdx);
105   setTile(rowIdx, colIdx, 0);
106 }
107
108 /***
109  * Gets the row index of the empty tile
110  *
111  * @return
112  */
113 public int getEmptyTileRow() {
114   return (container.indexOf(0) / size) + 1;
115 }
116
117 /***
118  * Gets the column index of the empty tile
119  *
120  * @return
121  */
122 public int getEmptyTileColumn() {
123   return (container.indexOf(0) - ((getEmptyTileRow() - 1) * size) + 1);
124 }
125
126 /***
127  * Gets the size of the board
128  *
129  * @return
130  */
131 public int size() {
132   return size;
133 }
134
135 /***
136  * Checks if the board is valid
137  *
138  * @return
139  */
140 public boolean isValid() {
141   if (container.stream().distinct().count() == size * size) {
142     for (int i = 0; i < (size * size) - 1; i++) {
143       if (!container.contains(i)) {
```

```
144           return false;
145         }
146       }
147       return true;
148     }
149     return false;
150   }
151
152   /***
153    * Shuffle the board
154    */
155   public void shuffle() {
156     Random rnd = new Random(System.nanoTime());
157     // Do 100 random moves
158     for (int i = 0; i < SHUFFLE_COUNT; i++) {
159       int random = rnd.nextInt(4);
160       try {
161         switch (random) {
162         case 0:
163           moveUp();
164           break;
165         case 1:
166           moveRight();
167           break;
168         case 2:
169           moveDown();
170           break;
171         case 3:
172           moveLeft();
173           break;
174         }
175       } catch (IllegalMoveException ex) {
176         // Ignore illegal moves
177       }
178     }
179   }
180
181   /***
182    * Moves the empty tile to the new position
183    *
184    * @param rowIdx
185    * @param colIdx
186    */
187   public void move(int rowIdx, int colIdx) throws IllegalMoveException {
188     if (rowIdx < 1 || rowIdx > size || colIdx < 1 || colIdx > size) {
189       throw new IllegalMoveException("Cannot move outside the board!");
190     }
191
192     int curRowIdx = getEmptyTileRow();
```

```
193     int curColIdx = getEmptyTileColumn();
194
195     if ((Math.abs(curRowIdx - rowIdx) == 1 && (curColIdx - colIdx == 0))
196         || (curRowIdx - rowIdx == 0 && Math.abs(curColIdx - colIdx) == 1)) {
197
198       int tile = getTile(rowIdx, colIdx);
199       setEmptyTile(rowIdx, colIdx);
200       setTile(curRowIdx, curColIdx, tile);
201     } else {
202       throw new IllegalMoveException("Cannot perform move!");
203     }
204
205   }
206
207   /***
208    * Moves the empty tile left
209    *
210    * @throws IllegalMoveException
211    */
212   public void moveLeft() throws IllegalMoveException {
213     int curRowIdx = getEmptyTileRow();
214     int curColIdx = getEmptyTileColumn();
215     move(curRowIdx, curColIdx - 1);
216   }
217
218   /***
219    * Moves the empty tile right
220    *
221    * @throws IllegalMoveException
222    */
223   public void moveRight() throws IllegalMoveException {
224     int curRowIdx = getEmptyTileRow();
225     int curColIdx = getEmptyTileColumn();
226     move(curRowIdx, curColIdx + 1);
227   }
228
229   /***
230    * Moves the empty tile up
231    *
232    * @throws IllegalMoveException
233    */
234   public void moveUp() throws IllegalMoveException {
235     int curRowIdx = getEmptyTileRow();
236     int curColIdx = getEmptyTileColumn();
237     move(curRowIdx - 1, curColIdx);
238   }
239
240   /***
241    * Moves the empty tile down
```

```
242     *
243     * @throws IllegalMoveException
244     */
245    public void moveDown() throws IllegalMoveException {
246      int curRowIdx = getEmptyTileRow();
247      int curColIdx = getEmptyTileColumn();
248      move(curRowIdx + 1, curColIdx);
249    }
250
251    /***
252     * Copies the board and all data
253     *
254     * @return
255     */
256    public Board copy() {
257      Board result = new Board(size);
258      result.container.clear();
259      result.container.addAll(container);
260      return result;
261    }
262
263    /***
264     * Execute the series of moves
265     *
266     * @param moves
267     * @throws IllegalMoveException
268     */
269    public void makeMoves(List<Move> moves) throws IllegalMoveException {
270      moves.forEach((x) -> {
271        move(x.getRow(), x.getCol());
272      });
273    }
274
275    /***
276     * clones the board
277     */
278    @Override
279    public Object clone() {
280      return this.copy();
281    }
282
283    /**
284     * Compares the size of this and the other board
285     */
286    @Override
287    public int compareTo(Board o) {
288      return size - o.size();
289    }
290
```

```
291   /***
292    * checks if the boards are equal
293    */
294   @Override
295   public boolean equals(Object other) {
296     if (this == other) {
297       return true;
298     }
299 
300     if (!(other instanceof Board)) {
301       return false;
302     }
303     Board otherBoard = (Board) other;
304     if (this.compareTo(otherBoard) != 0) {
305       return false;
306     }
307 
308     return this.container.equals(otherBoard.container);
309   }
310 
311   /***
312    * calculates the hashcode of the board
313    */
314   @Override
315   public int hashCode() {
316     final int prime = 31;
317     int result = 1;
318     result = prime * result
319         + ((container == null) ? 0 : container.hashCode());
320     return result;
321   }
322 
323   /***
324    * prints the board
325    */
326   @Override
327   public String toString() {
328     StringBuilder builder = new StringBuilder();
329     for (int i = 1; i <= size; i++) {
330       for (int j = 1; j <= size; j++) {
331         builder.append(String.format("%2d", getTile(i, j)));
332       }
333       builder.append("\n");
334     }
335     return builder.toString();
336   }
337 
338 }
```

Move.java

```java
package at.lumetsnet.puzzle;

public class Move {
  private int row;
  private int col;

  /**
   * @param row
   * @param col
   */
  public Move(int row, int col) {
    super();
    this.row = row;
    this.col = col;
  }

  /**
   * @return the row
   */
  public int getRow() {
    return row;
  }

  /**
   * @param row
   *            the row to set
   */
  public void setRow(int row) {
    this.row = row;
  }

  /**
   * @return the col
   */
  public int getCol() {
    return col;
  }

  /**
   * @param col
   *            the col to set
   */
  public void setCol(int col) {
    this.col = col;
  }

}
```

SearchNode.java

```
1  package at.lumetsnet.puzzle;
2
3  import java.util.ArrayList;
4  import java.util.Collections;
5  import java.util.List;
6
7  public class SearchNode implements Comparable<SearchNode> {
8
9    private Board board;
10   private SearchNode predecessor;
11   private int costsFromStart;
12   private Move move;
13   private int manhattanDistance;
14
15   /***
16    * Constructor with board
17    *
18    * @param board
19    */
20   public SearchNode(Board board) {
21     this.board = board;
22     // calculate manhattan distance
23     manhattanDistance = calcManhattanDistance();
24   }
25
26   /***
27    * Constructor with all data
28    *
29    * @param board
30    * @param predecessor
31    * @param costsFromStart
32    * @param move
33    */
34   public SearchNode(Board board, SearchNode predecessor, int costsFromStart,
35       Move move) {
36     this(board);
37     this.predecessor = predecessor;
38     this.costsFromStart = costsFromStart;
39     this.move = move;
40   }
41
42   /***
43    * Gets the board
44    */
45   public Board getBoard() {
```

```
46      return board;
47    }
48
49    /***
50     * Gets the predecessor
51     *
52     * @return
53     */
54    public SearchNode getPredecessor() {
55      return predecessor;
56    }
57
58    /***
59     * Sets the predecessor
60     *
61     * @param predecessor
62     */
63    public void setPredecessor(SearchNode predecessor) {
64      this.predecessor = predecessor;
65    }
66
67    /***
68     * Gets the costs from start
69     *
70     * @return
71     */
72    public int costsFromStart() {
73      return this.costsFromStart;
74    }
75
76    /***
77     * Sets the costs from start
78     *
79     * @param costsFromStart
80     */
81    public void setCostsFromStart(int costsFromStart) {
82      this.costsFromStart = costsFromStart;
83    }
84
85    /***
86     * Gets the estimated costs to the goal
87     *
88     * @return
89     */
90    public int estimatedCostsToTarget() {
91      return manhattanDistance;
92    }
93
94    /***
```

```
95    * Gets the estimated total costs
96    *
97    * @return
98    */
99   public int estimatedTotalCosts() {
100    return costsFromStart + estimatedCostsToTarget();
101  }
102
103  /***
104   * Gets the move represented by this node
105   *
106   * @return
107   */
108  public Move getMove() {
109    return move;
110  }
111
112  /***
113   * Sets the move represented by this node
114   *
115   * @return
116   */
117  public void setMove(Move move) {
118    this.move = move;
119  }
120
121  /***
122   * Compares the node with another object
123   */
124  public boolean equals(Object other) {
125    if (other == null)
126      return false;
127    if (!(other instanceof SearchNode))
128      return false;
129
130    SearchNode otherNode = (SearchNode) other;
131    if (board == null && otherNode.board != null)
132      return false;
133    return board.equals(otherNode.board);
134  }
135
136  /***
137   * Compare the costs of the node against the other
138   *
139   * @return
140   */
141  @Override
142  public int compareTo(SearchNode other) {
143    return estimatedTotalCosts() - other.estimatedTotalCosts();
```

```
144   }
145
146   /***
147    * Converts the list into a list of moves from the start
148    *
149    * @return
150    */
151   public List<Move> toMoves() {
152     List<Move> result = new ArrayList<Move>();
153     SearchNode cur = this;
154     while (cur != null) {
155       if (cur.getMove() != null) {
156         result.add(cur.getMove());
157       }
158       cur = cur.getPredecessor();
159     }
160     // reverse the order of the collection
161     // to get the moves from the start
162     Collections.reverse(result);
163     return result;
164   }
165
166   /***
167    * Calculates the manhattan distance
168    *
169    * @return
170    */
171   private int calcManhattanDistance() {
172     int manhattanDistanceSum = 0;
173     for (int x = 1; x <= board.size(); x++)
174       for (int y = 1; y <= board.size(); y++) {
175         int value = board.getTile(x, y);
176         if (value != 0) {
177           int targetX = (value - 1) / board.size();
178           int targetY = (value - 1) % board.size();
179           int dx = x - (targetX + 1);
180           int dy = y - (targetY + 1);
181           manhattanDistanceSum += Math.abs(dx) + Math.abs(dy);
182         }
183       }
184     return manhattanDistanceSum;
185   }
186
187
188   /***
189    * calculates the hashcode of the SearchNode
190    */
191   @Override
192   public int hashCode() {
```

```
193     final int prime = 17;
194     int result = 1;
195     result = prime * result
196         + ((board == null) ? 0 : board.hashCode());
197     return result;
198   }
199
200
201
202
203
204 }
```

SlidingPuzzle.java

```
1  package at.lumetsnet.puzzle;
2
3  import java.util.ArrayList;
4  import java.util.HashSet;
5  import java.util.List;
6  import java.util.PriorityQueue;
7  import java.util.Queue;
8
9  import at.lumetsnet.puzzle.exceptions.IllegalMoveException;
10 import at.lumetsnet.puzzle.exceptions.NoSolutionException;
11
12 public class SlidingPuzzle {
13
14   /***
15    * Solves the board
16    *
17    * @param board
18    * @return
19    */
20   public List<Move> solve(Board board) {
21     Queue<SearchNode> openQueue = new PriorityQueue<SearchNode>();
22     HashSet<SearchNode> closedSet = new HashSet<SearchNode>();
23
24     // create search node from current board
25     SearchNode current = new SearchNode(board);
26     openQueue.add(current);
27
28     while (!openQueue.isEmpty()) {
29       // get next node
30       current = openQueue.poll();
31
32       // estimatedCostsToTarget = 0 means we found a solution
33       if (current.estimatedCostsToTarget() == 0) {
34         return current.toMoves();
```

```
35        }
36
37        closedSet.add(current);
38
39        // calculate the successors
40        final List<SearchNode> successors = getSuccessors(current);
41        for (SearchNode successor : successors) {
42
43          if (!closedSet.contains(successor)) {
44            if (openQueue.contains(successor)
45                && current.estimatedTotalCosts() >= successor
46                    .estimatedTotalCosts()) {
47              //remove old node
48              openQueue.remove(successor);
49            }
50            openQueue.add(successor);
51          }
52        }
53      }
54      throw new NoSolutionException("Board has no solution");
55    }
56
57    /***
58     * returns the successors for the node
59     *
60     * @param parent
61     * @return
62     */
63    private List<SearchNode> getSuccessors(SearchNode parent) {
64      final List<SearchNode> result = new ArrayList<SearchNode>();
65      for (int i = 0; i < 4; i++) {
66        Board newBoard = parent.getBoard().copy();
67        try {
68          switch (i) {
69          case 0:
70            newBoard.moveLeft();
71            break;
72          case 1:
73            newBoard.moveUp();
74            break;
75          case 2:
76            newBoard.moveRight();
77            break;
78          case 3:
79            newBoard.moveDown();
80            break;
81          }
82          SearchNode node = new SearchNode(newBoard, parent,
83              parent.costsFromStart() + 1, new Move(
```

```
84                  newBoard.getEmptyTileRow(),
85                  newBoard.getEmptyTileColumn()));
86          result.add(node);
87        } catch (IllegalMoveException ex) {
88          // ignore illegal moves
89        }
90      }
91      return result;
92    }
93
94    /***
95     * Prints the moves to the console
96     *
97     * @param board
98     * @param moves
99     */
100   public void printMoves(Board board, List<Move> moves) {
101     System.out.println("Starting board");
102     System.out.println(board);
103     moves.stream().forEach((x) -> {
104       board.move(x.getRow(), x.getCol());
105       System.out.println(board);
106     });
107   }
108 }
```

AbstractTest.java

```
1  package at.lumetsnet.puzzle.tests;
2
3  import static org.junit.Assert.assertTrue;
4  import at.lumetsnet.puzzle.Board;
5
6  public class AbstractTest {
7    /***
8     * Creates a 3x3 testboard with empty tile on 2x2
9     *
10    * @return board
11    */
12   protected Board getTestBoard() {
13     Board board = new Board(3);
14     board.setTile(1, 1, 1);
15     board.setTile(1, 2, 2);
16     board.setTile(1, 3, 3);
17     board.setTile(2, 1, 4);
18     board.setEmptyTile(2, 2);
19     board.setTile(2, 3, 6);
20     board.setTile(3, 1, 7);
21     board.setTile(3, 2, 8);
```

```
22      board.setTile(3, 3, 5);
23
24      assertTrue(board.isValid());
25      return board;
26    }
27  }
```

BoardTest.java

```
1  package at.lumetsnet.puzzle.tests;
2
3  import static org.junit.Assert.*;
4
5  import java.util.ArrayList;
6  import java.util.List;
7  import java.util.logging.Logger;
8
9  import org.junit.Test;
10 import org.junit.experimental.categories.Categories.ExcludeCategory;
11
12 import at.lumetsnet.puzzle.Board;
13 import at.lumetsnet.puzzle.Move;
14 import at.lumetsnet.puzzle.exceptions.BoardException;
15 import at.lumetsnet.puzzle.exceptions.IllegalMoveException;
16
17 public class BoardTest extends AbstractTest {
18
19   @Test
20   public void getTileTest() {
21     Board board = getTestBoard();
22     assertEquals(0, board.getTile(2, 2));
23     assertEquals(1, board.getTile(1, 1));
24   }
25
26   @Test
27   public void setTileTest() {
28     Board board = new Board(3);
29     // default empty tile
30     assertEquals(1, board.getTile(1, 1));
31     board.setTile(1, 1, 2);
32     assertEquals(2, board.getTile(1, 1));
33   }
34
35   @Test
36   public void getEmptyTileTest() {
37     Board board = getTestBoard();
38     assertEquals(2, board.getEmptyTileColumn());
39     assertEquals(2, board.getEmptyTileRow());
40   }
```

```
41
42  public void sizeTest() {
43    Board board = getTestBoard();
44    assertEquals(2, board.size());
45    board = new Board(9);
46    assertEquals(9, board.size());
47  }
48
49  @Test
50  public void simpleIsValidTest() {
51    Board board;
52    try {
53      board = getTestBoard();
54      assertTrue(board.isValid());
55    } catch (BoardException e) {
56      fail("BoardException not expected.");
57    }
58  }
59
60  @Test
61  public void simpleIsNotValidTest() {
62    Board board;
63    try {
64      board = new Board(3);
65      board.setTile(1, 1, 1);
66      board.setTile(1, 2, 2);
67      board.setTile(1, 3, 3);
68      board.setTile(2, 1, 4);
69      board.setTile(2, 2, 5);
70      board.setTile(2, 3, 6);
71      board.setTile(3, 1, 7);
72      board.setTile(3, 2, 1);
73      board.setTile(3, 3, 0);
74
75      assertTrue(!board.isValid());
76    } catch (BoardException e) {
77      fail("BoardException not expected.");
78    }
79  }
80
81  @Test
82  public void simpleIsNotValidTest2() {
83    Board board;
84    try {
85      board = new Board(3);
86      board.setTile(1, 1, 8);
87      board.setTile(1, 2, 2);
88      board.setTile(1, 3, 0);
89      board.setTile(2, 1, 7);
```

```
90        board.setTile(2, 2, 5);
91        board.setTile(2, 3, 4);
92        board.setTile(3, 1, 3);
93        board.setTile(3, 2, 1);
94        board.setTile(3, 3, 6);
95
96        assertTrue(board.isValid());
97      } catch (BoardException e) {
98        fail("BoardException not expected.");
99      }
100   }
101
102   @Test
103   public void simpleIsNotValidTest3() {
104     Board board;
105     try {
106       board = new Board(3);
107       board.setTile(1, 1, 8);
108       board.setTile(1, 2, 2);
109       // not all tiles set
110       assertFalse(board.isValid());
111     } catch (BoardException e) {
112       fail("BoardException not expected.");
113     }
114   }
115
116   @Test
117   public void simpleIsNotValidTest4() {
118     Board board;
119     try {
120       board = new Board(3);
121       board.setTile(1, 1, 8);
122       board.setTile(1, 2, 2);
123       board.setTile(1, 3, 0);
124       board.setTile(2, 1, 7);
125       board.setTile(2, 2, 5);
126       board.setTile(2, 3, 4);
127       board.setTile(3, 1, 3);
128       board.setTile(3, 2, 1);
129       board.setTile(3, 3, 8); // invalid entry
130
131       assertFalse(board.isValid());
132     } catch (BoardException e) {
133       fail("BoardException not expected.");
134     }
135   }
136
137   @Test
138   public void moveOutsideTest() {
```

```
139     Board board = getTestBoard();
140     boolean hadExeption = false;
141     try {
142       board.move(1, 0);
143     } catch (IllegalMoveException ex) {
144       hadExeption = true;
145     }
146     assertTrue(hadExeption);
147     hadExeption = false;
148     try {
149       board.move(0, 1);
150     } catch (IllegalMoveException ex) {
151       hadExeption = true;
152     }
153     assertTrue(hadExeption);
154     hadExeption = false;
155     try {
156       board.move(4, 1);
157     } catch (IllegalMoveException ex) {
158       hadExeption = true;
159     }
160     assertTrue(hadExeption);
161     hadExeption = false;
162     try {
163       board.move(1, 4);
164     } catch (IllegalMoveException ex) {
165       hadExeption = true;
166     }
167     assertTrue(hadExeption);
168     hadExeption = false;
169 
170   }
171 
172   @Test
173   public void illegalMoveTest() {
174     Board board = getTestBoard();
175     boolean hadExeption = false;
176     try {
177       board.move(1, 3);
178     } catch (IllegalMoveException ex) {
179       hadExeption = true;
180     }
181     assertTrue(hadExeption);
182     hadExeption = false;
183 
184     try {
185       board.move(3, 1);
186     } catch (IllegalMoveException ex) {
187       hadExeption = true;
```

```
188     }
189     assertTrue(hadExeption);
190     hadExeption = false;
191     try {
192       board.move(2, 2);
193     } catch (IllegalMoveException ex) {
194       hadExeption = true;
195     }
196     assertTrue(hadExeption);
197     hadExeption = false;
198   }
199
200   @Test
201   public void allowdMovesTest() {
202     Board board = getTestBoard();
203     board.move(2, 3);
204     assertTrue(board.isValid());
205     assertEquals(2, board.getEmptyTileRow());
206     assertEquals(3, board.getEmptyTileColumn());
207
208     board = getTestBoard();
209     board.move(3, 2);
210     assertTrue(board.isValid());
211     assertEquals(3, board.getEmptyTileRow());
212     assertEquals(2, board.getEmptyTileColumn());
213
214     board = getTestBoard();
215     board.move(2, 1);
216     assertTrue(board.isValid());
217     assertEquals(2, board.getEmptyTileRow());
218     assertEquals(1, board.getEmptyTileColumn());
219
220     board = getTestBoard();
221     board.move(2, 3);
222     assertTrue(board.isValid());
223     assertEquals(2, board.getEmptyTileRow());
224     assertEquals(3, board.getEmptyTileColumn());
225   }
226
227   @Test
228   public void moveLeftTest() {
229     Board board = getTestBoard();
230     board.moveLeft();
231     assertTrue(board.isValid());
232     assertEquals(2, board.getEmptyTileRow());
233     assertEquals(1, board.getEmptyTileColumn());
234   }
235
236   @Test(expected = IllegalMoveException.class)
```

```
237   public void moveLeftExceptionTest() {
238     Board board = getTestBoard();
239     board.moveLeft();
240     board.moveLeft();
241   }
242
243   @Test
244   public void moveRightTest() {
245     Board board = getTestBoard();
246     board.moveRight();
247     assertTrue(board.isValid());
248     assertEquals(2, board.getEmptyTileRow());
249     assertEquals(3, board.getEmptyTileColumn());
250   }
251
252   @Test(expected = IllegalMoveException.class)
253   public void moveRightExceptionTest() {
254     Board board = getTestBoard();
255     board.moveRight();
256     board.moveRight();
257   }
258
259   @Test
260   public void moveUpTest() {
261     Board board = getTestBoard();
262     board.moveUp();
263     assertTrue(board.isValid());
264     assertEquals(1, board.getEmptyTileRow());
265     assertEquals(2, board.getEmptyTileColumn());
266   }
267
268   @Test(expected = IllegalMoveException.class)
269   public void moveUpExceptionTest() {
270     Board board = getTestBoard();
271     board.moveUp();
272     board.moveUp();
273   }
274
275   @Test
276   public void moveDownTest() {
277     Board board = getTestBoard();
278     board.moveDown();
279     assertTrue(board.isValid());
280     assertEquals(3, board.getEmptyTileRow());
281     assertEquals(2, board.getEmptyTileColumn());
282   }
283
284   @Test(expected = IllegalMoveException.class)
285   public void moveDownExceptionTest() {
```

```
286     Board board = getTestBoard();
287     board.moveDown();
288     board.moveDown();
289   }
290
291   @Test
292   public void equalsTest() {
293     Board testBoard = getTestBoard();
294     Board sameBoard = getTestBoard();
295     Board otherBoard = getTestBoard();
296     otherBoard.moveDown();
297
298     assertTrue(testBoard.equals(sameBoard));
299     assertFalse(testBoard.equals(otherBoard));
300     assertFalse(testBoard.equals(1));
301   }
302
303   @Test
304   public void compareToTest() {
305     Board testBoard = new Board(3);
306     Board sameSizeBoard = new Board(3);
307     Board biggerBoard = new Board(4);
308     Board smallerBoard = new Board(2);
309     assertTrue(testBoard.compareTo(sameSizeBoard) == 0);
310     assertTrue(testBoard.compareTo(smallerBoard) == 1);
311     assertTrue(testBoard.compareTo(biggerBoard) == -1);
312   }
313
314   @Test
315   public void copyTest() {
316     Board board = getTestBoard();
317     Board copyBoard = board.copy();
318     // change original board to check copy of references
319     board.moveDown();
320     Board originalBoard = getTestBoard();
321     // check against original board
322     assertTrue(copyBoard.equals(originalBoard));
323     // check against changed board
324     assertFalse(copyBoard.equals(board));
325   }
326
327   @Test
328   public void shuffleTest() {
329     Board board = getTestBoard();
330     board.shuffle();
331     // check that the board is not the same as the
332     // original board
333     assertTrue(board.isValid());
334     assertFalse(board.equals(getTestBoard()));
```

```
335
336   }
337
338   @Test
339   public void makeMovesTest() {
340     Board board = getTestBoard();
341     List<Move> moveList = new ArrayList<Move>();
342     moveList.add(new Move(1, 2));
343     moveList.add(new Move(1, 3));
344     moveList.add(new Move(2, 3));
345     moveList.add(new Move(3, 3));
346     moveList.add(new Move(3, 2));
347     board.makeMoves(moveList);
348     assertTrue(board.isValid());
349     assertEquals(3, board.getEmptyTileRow());
350     assertEquals(2, board.getEmptyTileColumn());
351   }
352 }
```

SearchNodeTest.java

```
1  package at.lumetsnet.puzzle.tests;
2
3  import static org.junit.Assert.*;
4
5  import org.junit.Test;
6
7  import at.lumetsnet.puzzle.Board;
8  import at.lumetsnet.puzzle.SearchNode;
9  import at.lumetsnet.puzzle.exceptions.BoardException;
10
11 public class SearchNodeTest extends AbstractTest {
12
13   @Test
14   public void estimatedCostsToTargetTest() {
15     Board board = getTestBoard();
16     SearchNode node = new SearchNode(board);
17     assertEquals(2, node.estimatedCostsToTarget());
18   }
19
20   @Test
21   public void simpleNodeTest() {
22     try {
23       Board board = new Board(3);
24       board.setTile(1, 1, 1);
25       board.setTile(1, 2, 2);
26       board.setTile(1, 3, 3);
27       board.setTile(2, 1, 4);
28       board.setTile(2, 2, 5);
```

```
29        board.setTile(2, 3, 6);
30        board.setTile(3, 1, 7);
31        board.setTile(3, 2, 8);
32        board.setTile(3, 3, 0);
33        SearchNode node = new SearchNode(board);
34        assertEquals(0, node.estimatedCostsToTarget());
35
36        board = new Board(3);
37        board.setTile(1, 1, 1);
38        board.setTile(1, 2, 2);
39        board.setTile(1, 3, 3);
40        board.setTile(2, 1, 4);
41        board.setTile(2, 2, 0);
42        board.setTile(2, 3, 6);
43        board.setTile(3, 1, 7);
44        board.setTile(3, 2, 8);
45        board.setTile(3, 3, 5);
46        node = new SearchNode(board);
47        assertEquals(2, node.estimatedCostsToTarget());
48
49        board = new Board(3);
50        board.setTile(1, 1, 1);
51        board.setTile(1, 2, 0);
52        board.setTile(1, 3, 3);
53        board.setTile(2, 1, 4);
54        board.setTile(2, 2, 5);
55        board.setTile(2, 3, 6);
56        board.setTile(3, 1, 7);
57        board.setTile(3, 2, 8);
58        board.setTile(3, 3, 2);
59        node = new SearchNode(board);
60        assertEquals(3, node.estimatedCostsToTarget());
61      } catch (BoardException e) {
62        fail("Unexpeced BoardException.");
63      }
64    }
65
66  }
```

SlidingPuzzleSolverTest.java

```
1  package at.lumetsnet.puzzle.tests;
2
3  import static org.junit.Assert.*;
4
5  import java.util.List;
6
7  import org.junit.Test;
8
```

```
9  import at.lumetsnet.puzzle.Board;
10 import at.lumetsnet.puzzle.Move;
11 import at.lumetsnet.puzzle.SlidingPuzzle;
12 import at.lumetsnet.puzzle.exceptions.NoSolutionException;
13
14 public class SlidingPuzzleSolverTest {
15
16   @Test
17   public void solveSimplePuzzleTest1() {
18     try {
19       SlidingPuzzle solver = new SlidingPuzzle();
20       Board board = new Board(3);
21       board.setTile(1, 1, 1);
22       board.setTile(1, 2, 2);
23       board.setTile(1, 3, 3);
24       board.setTile(2, 1, 4);
25       board.setTile(2, 2, 5);
26       board.setTile(2, 3, 6);
27       board.setTile(3, 1, 7);
28       board.setTile(3, 2, 0);
29       board.setTile(3, 3, 8);
30
31       List<Move> moves = solver.solve(board);
32       assertEquals(1, moves.size());
33       assertTrue(moves.get(0).getRow() == 3 && moves.get(0).getCol() == 3);
34     } catch (NoSolutionException nse) {
35       fail("NoSolutionException is not expected.");
36     }
37   }
38
39   @Test
40   public void solveSimplePuzzleTest2() {
41     try {
42       SlidingPuzzle solver = new SlidingPuzzle();
43       Board board = new Board(3);
44       board.setTile(1, 1, 1);
45       board.setTile(1, 2, 2);
46       board.setTile(1, 3, 3);
47       board.setTile(2, 1, 4);
48       board.setTile(2, 2, 5);
49       board.setTile(2, 3, 6);
50       board.setTile(3, 1, 0);
51       board.setTile(3, 2, 7);
52       board.setTile(3, 3, 8);
53
54       List<Move> moves = solver.solve(board);
55       assertEquals(2, moves.size());
56       assertTrue(moves.get(0).getRow() == 3 && moves.get(0).getCol() == 2);
57       assertTrue(moves.get(1).getRow() == 3 && moves.get(1).getCol() == 3);
```

```
58      } catch (NoSolutionException nse) {
59        fail("NoSolutionException is not expected.");
60      }
61    }
62
63    @Test
64    public void solveComplexPuzzleTest1() {
65
66      try {
67        SlidingPuzzle solver = new SlidingPuzzle();
68
69        // 8 2 7
70        // 1 4 6
71        // 3 5 X
72        Board board = new Board(3);
73        board.setTile(1, 1, 8);
74        board.setTile(1, 2, 2);
75        board.setTile(1, 3, 7);
76        board.setTile(2, 1, 1);
77        board.setTile(2, 2, 4);
78        board.setTile(2, 3, 6);
79        board.setTile(3, 1, 3);
80        board.setTile(3, 2, 5);
81        board.setTile(3, 3, 0);
82
83        Board copy = board.copy();
84        List<Move> moves = solver.solve(board);
85        System.out.println(moves.size());
86        board.makeMoves(moves);
87        assertEquals(new Board(3), board);
88        solver.printMoves(copy, moves);
89      } catch (NoSolutionException nse) {
90        fail("NoSolutionException is not expected.");
91      }
92    }
93
94    @Test
95    public void solveRandomPuzzlesTest() {
96      SlidingPuzzle solver = new SlidingPuzzle();
97
98      for (int k = 0; k < 50; k++) {
99        try {
100         Board board = new Board(3);
101         int n = 1;
102         int maxN = board.size() * board.size();
103         for (int i = 1; i <= board.size(); i++)
104           for (int j = 1; j <= board.size(); j++)
105             board.setTile(i, j, (n++) % maxN);
106
```

```
107         board.shuffle();
108
109         List<Move> moves = solver.solve(board);
110         board.makeMoves(moves);
111         assertEquals(new Board(3), board);
112       } catch (NoSolutionException nse) {
113         fail("NoSolutionException is not expected.");
114       }
115     }
116   }
117
118   @Test
119   public void solveSimplePuzzleTest_4x4() {
120     try {
121       SlidingPuzzle solver = new SlidingPuzzle();
122       Board board = new Board(4);
123
124       board.moveLeft();
125
126       List<Move> moves = solver.solve(board);
127       assertEquals(1, moves.size());
128       assertTrue(moves.get(0).getRow() == 4 && moves.get(0).getCol() == 4);
129     } catch (NoSolutionException nse) {
130       fail("NoSolutionException is not expected.");
131     }
132   }
133
134   @Test
135   public void solveComplexPuzzleTest_4x4() {
136     try {
137       SlidingPuzzle solver = new SlidingPuzzle();
138       Board board = new Board(4);
139
140       board.moveLeft();
141       board.moveLeft();
142       board.moveUp();
143       board.moveLeft();
144       board.moveUp();
145       board.moveUp();
146       board.moveRight();
147       board.moveDown();
148       board.moveLeft();
149
150       List<Move> moves = solver.solve(board);
151       board.makeMoves(moves);
152       assertEquals(new Board(4), board);
153     } catch (NoSolutionException nse) {
154       fail("NoSolutionException is not expected.");
155     }
```

```
156   }
157
158 }
```

BoardException.java

```
1  package at.lumetsnet.puzzle.exceptions;
2
3  public class BoardException extends RuntimeException {
4
5    /**
6     * Serial Id
7     */
8    private static final long serialVersionUID = 1L;
9
10   public BoardException(String message) {
11     super(message);
12   }
13 }
```

IllegalMoveException.java

```
1  package at.lumetsnet.puzzle.exceptions;
2
3  public class IllegalMoveException extends BoardException {
4    /**
5     * Serial Id
6     */
7    private static final long serialVersionUID = 1L;
8
9    public IllegalMoveException(String message) {
10     super(message);
11   }
12 }
```

InvalidBoardIndexException.java

```
1  package at.lumetsnet.puzzle.exceptions;
2
3  public class InvalidBoardIndexException extends BoardException {
4
5    /**
6     * Serial Id
7     */
8    private static final long serialVersionUID = 1L;
9
10   public InvalidBoardIndexException(String message) {
```

```
11      super(message);
12    }
13  }
```

InvalidTileNumberException.java

```
1  package at.lumetsnet.puzzle.exceptions;
2
3  public class InvalidTileNumberException extends BoardException {
4
5    /**
6     * Serial Id
7     */
8    private static final long serialVersionUID = 1L;
9
10   public InvalidTileNumberException(String message) {
11     super(message);
12   }
13 }
```

NoSolutionException.java

```
1  package at.lumetsnet.puzzle.exceptions;
2
3  public class NoSolutionException extends RuntimeException {
4    /**
5     * Serial Id
6     */
7    private static final long serialVersionUID = 1L;
8
9    public NoSolutionException(String message) {
10     super(message);
11   }
12 }
```

1.3 Testfälle