Praktische Informatik II
Tutor:in: Tobias Diehl

SoSe 2024
Bearbeiter:in: Altug Uyanik, Öykü Koç

Übungsblatt 2

Lösungsvorschlag

InsectHotel.java

Die InsectHotel Klasse wurde von **Öykü Koç** verfasst.

Dieser Code enthält eine Java-Enum (Aufzählung) Definition. Es wurde eine Enum namens ‘Result‘ definiert, die drei Konstanten enthält: ‘INVALID‘, ‘VALID‘ und ‘SOLVED‘. ‘INVALID‘ gibt an, dass die aktuelle teilweise Belegung des Hotels ungültig ist. Das bedeutet, dass die aktuellen Reservierungen oder die Belegung des Hotels möglicherweise nicht den geltenden Regeln entsprechen. ‘VALID‘ zeigt an, dass die aktuelle teilweise Belegung des Hotels gültig ist. Das bedeutet, dass die aktuellen Reservierungen oder die Belegung des Hotels den geltenden Regeln entsprechen. ‘SOLVED‘ gibt an, dass das Hotel vollständig belegt und gültig ist. Das bedeutet, dass alle Zimmer des Hotels reserviert sind und die Belegung den geltenden Regeln entspricht.

```
1  package de.uni_bremen.pi2;
2
3  // INVALID, VALID und SOLVED können direkt verwendet werden.
4
5  import java.util.Arrays;
6
7  import static de.uni_bremen.pi2.InsectHotel.Result.*;
8
9  /**
10  * Author Öykü Koç
11  */
12 public class InsectHotel {
13     /**
14      * Die drei möglichen Ergebnisse der Methode {@link #check}.
15      * INVALID: Die aktuelle Teilbelegung des Hotels ist bereits ungültig.
16      * VALID: Die aktuelle Teilbelegung des Hotels ist gültig.
17      * SOLVED: Das Hotel ist voll belegt und gültig.
18      */
19     enum Result {INVALID, VALID, SOLVED}
```

Aufgabe 1.1 Ausfüllen

Die Methode ‘fill‘ erhält das eigentliche Rätsel in Form eines rechteckigen String-Arrays, in dem bereits einige Kästchen mit X oder O belegt sind. Die anderen sind leer (Leerzeichen). Zusätzlich erhält die Methode eine Abfolge von X und O, mit der sie die freien Plätze des Rätsels ausfüllen muss. Das Ausfüllen erfolgt in einem neuen String-Array, d. h. die Parameter von ‘fill‘ werden nicht verändert. Das Ausfüllen erfolgt immer von oben links nach unten rechts (zeilenweise). Wenn ein Kästchen bereits im Rätsel belegt ist, wird sein Inhalt übernommen. Ist es hingegen frei, wird der nächste Eintrag aus der übergebenen Folge verwendet. Die Folge kann kürzer sein als die Anzahl der freien Plätze. Wenn es keine Einträge mehr in der Folge gibt, werden ab dann immer die Kästchen aus dem Rätsel übernommen, d. h. auch leere Kästchen. ‘fill‘ gibt das (teil-)ausgefüllte Rätsel als Ergebnis zurück.

```
20     /**
21      * Die Methode füllt ein zum Teil belegtes Hotel mit weiteren
22      * Einträgen auf. Dabei werden die weiteren Einträge der Reihe nach
23      * von oben links nach unten rechts eingetragen (horizontal zuerst).
24      * Dabei bleiben die existierenden Einträge erhalten und die neuen
```

```
25      * Einträge ersetzen nur die Lücken (' ').
26      *
27      * @param hotel   Das zum Teil belegte Hotel. Wird nicht verändert.
28      *                {@see #solve}.
29      * @param entries Eine Folge aus 'O' und 'X', die in die Lücken in
30      *                {@code hotel} eintragen wird.
31      * @return Ein Hotel, bei dem {@code entries} in die Lücken von
32      * {@code hotel} eingesetzt wurde, zumindest so weit, wie
33      * {@code entries} gereicht hat. Da {@code hotel} nicht geändert
34      * werden darf, ist dies ein neues Objekt.
35      */
36     static String[] fill(final String[] hotel, final String entries) {
37
38         final StringBuilder sb = new StringBuilder();
39
40         for (String value : hotel) {
41             sb.append(value);
42         }
43
44         for (char character : entries.toCharArray()) {
45             final int position = sb.indexOf(" ");
46             sb.setCharAt(position, character);
47         }
48
49         final int lengthOfRow = hotel[0].length();
50         for (int i = 0; i < hotel.length; i++) {
51             hotel[i] = sb.substring(i * lengthOfRow, i * lengthOfRow + lengthOfRow);
52         }
53
54         return hotel;
55     }
```

Aufgabe 1.2 Prüfen

Die erste Bedingung überprüft, ob die Länge des Arrays hotel gleich Null ist. Wenn die Länge des Arrays Null ist, was bedeutet, dass es keine Elemente enthält, wird eine 'IllegalArgumentException'-Ausnahme ausgelöst. Diese Ausnahme wird mit der Nachricht "`String-Array-Parameter kann nicht leer sein!`" geworfen, was bedeutet, dass der String-Array-Parameter nicht leer sein kann.

```
57     /**
58      * Die Methode überprüft, ob ein Hotel gültig (teil-) belegt ist.
59      * Es dürfen horizontal und vertikal maximal jeweils zwei gleiche
60      * Einträge ('O' bzw. 'X') aufeinander folgen. In jeder voll
61      * ausgefüllten Zeile und Spalte müssen gleich viele Einträge beider
62      * Sorten stehen.
63      *
64      * @param hotel Das zum Teil belegte Hotel. {@see #solve}.
65      * @return INVALID: Wenn eine der Anforderungen nicht erfüllt ist - oder
66      * besser - auch mit späteren Eintragungen nicht mehr erfüllt
67      * werden kann.
68      * VALID: Bisher ist keine Anforderung verletzt und mit weiteren
69      * Eintragungen könnte noch eine gültige Lösung entstehen.
70      * SOLVED: Das Hotel ist vollständig belegt und keine der
71      * Anforderungen ist verletzt.
72      */
73     static Result check(final String[] hotel) {
74         int X_count;
75         int O_count;
76         int blank_Count = 0;
77         String colValue = "";
78
79         // GIBT EINE FEHLERMELDUNG AUS, WENN ER NULL ODER LEER IST
80         if (hotel.length == 0)
81             throw new IllegalArgumentException("String-Array-Parameter kann nicht leer
                    sein!");
```

Die zweite Bedingung überprüft, ob das Array hotel Null ist. Wenn das Array hotel Null ist, was bedeutet, dass es keine Referenz enthält, wird erneut eine ‘IllegalArgumentException‘-Ausnahme ausgelöst. Diese Ausnahme wird mit der Nachricht “`String-Array-Parameter kann keinen Nullwert annehmen!`” geworfen, was bedeutet, dass der String-Array-Parameter keinen Nullwert akzeptieren kann.

```
83          if (hotel == null)
84              throw new IllegalArgumentException("String-Array-Parameter kann keinen
                    Nullwert annehmen!");
```

Das Ziel dieser Methode ist es, die Gültigkeit einer Hotelanordnung zu überprüfen. Das Hotel wird als Matrix dargestellt, die teilweise mit den Zeichen ’X’ und ’O’ gefüllt ist. Die Methode überprüft die Gültigkeit der Hotelanordnung gemäß bestimmten Regeln.

Horizontale Anordnungskontrolle:Die Methode überprüft die horizontale Anordnung des Hotels (Zeilen). Wenn in jeder Zeile drei aufeinanderfolgende ’X’- oder ’O’-Zeichen vorhanden sind, wird die Hotelanordnung als ungültig betrachtet und “`INVALID`” zurückgegeben. Außerdem wird die Anzahl der ’X’- und ’O’-Zeichen in jeder Zeile überprüft, und wenn eine bestimmte Grenze überschritten wird, wird die Anordnung ebenfalls als ungültig betrachtet und “`INVALID`” zurückgegeben.

Vertikale Anordnungskontrolle: Die Methode überprüft die vertikale Anordnung des Hotels (Spalten). Wenn in jeder Spalte drei aufeinanderfolgende ’X’- oder ’O’-Zeichen vorhanden sind, wird die Hotelanordnung als ungültig betrachtet und “`INVALID`” zurückgegeben. Außerdem wird die Anzahl der ’X’- und ’O’-Zeichen in jeder Spalte überprüft, und wenn eine bestimmte Grenze überschritten wird, wird die Anordnung ebenfalls als ungültig betrachtet und “`INVALID`” zurückgegeben.

Bestimmung der Ergebnisse:Wenn alle leeren Felder im Hotel ausgefüllt sind und keine Ungültigkeiten festgestellt wurden, wird die Hotelanordnung als gültig betrachtet und “"VALID“” zurückgegeben. Wenn das Hotel vollständig gefüllt ist und keine Ungültigkeiten festgestellt wurden, wird die Hotelanordnung als gelöst betrachtet und “"SOLVED“” zurückgegeben.

Diese Methode analysiert die Gültigkeit der Hotelanordnung anhand horizontaler und vertikaler Anordnungen und bewertet die Ergebnisse gemäß bestimmten Regeln. Dadurch liefert sie Informationen über die Gültigkeit der Hotelanordnung und gibt bei Bedarf ein entsprechendes Ergebnis zurück.

```
87          // HORIZONTALE KONTROLLE (VON LINKS NACH RECHTS) -> UNGÜLTIGE STEUERUNG.
88          for (String row : hotel) {
89
90              if (row.contains("XXX") || row.contains("OOO"))
91                  return Result.INVALID;
92
93              X_count = 0;
94              O_count = 0;
95              blank_Count = 0;
96
97              for (char character : row.toCharArray()) {
98                  if (character == 'X')
99                      X_count++;
100                 else if (character == 'O')
101                     O_count++;
102                 else if (character == ' ')
103                     blank_Count++;
104             }
105
106             if (blank_Count == 0) {
107                 if (X_count != O_count)
108                     return Result.INVALID;
109             }
110         }
111
112         // VERTIKALE STEUERUNG (VON OBEN NACH UNTEN) -> UNGÜLTIGE STEUERUNG.
113         for (int i = 0; i < hotel.length; i++) {
```

```
114             X_count = 0;
115             O_count = 0;
116             blank_Count = 0;
117 
118             for (String row : hotel) {
119                 colValue += row.charAt(i);
120 
121                 if (colValue.toString().contains("XXX") || colValue.toString().contains("
                        OOO")) {
122                     return Result.INVALID;
123                 }
124 
125                 if (row.charAt(i) == 'X')
126                     X_count++;
127                 else if (row.charAt(i) == 'O')
128                     O_count++;
129                 else if (row.charAt(i) == ' ')
130                     blank_Count++;
131             }
132 
133             if (blank_Count == 0) {
134                 if (X_count != O_count)
135                     return Result.INVALID;
136             }
137         }
138 
139         // GÜLTIGE STEUERUNG.
140         if (blank_Count > 0) {
141             return Result.VALID;
142         }
143 
144         return Result.SOLVED;
145     }
```

Aufgabe 1.3 Lösen

Nach den letzten beiden Aufgaben ist es nun möglich, eine Abfolge von X- und O-Zeichen in das Insektenhotel einzufügen und dann zu überprüfen, ob die resultierende Belegung gültig ist und möglicherweise eine Lösung darstellt. In dieser Aufgabe wird nun die eigentliche Suche nach der Lösung in der Methode solve implementiert. Dazu genügt es, analog zum Unterricht, eine immer längere Abfolge zu erzeugen.
Zunächst wird der Array-Parameter "hotel"überprüft, ob er null ist. - Wenn "hotel"null ist, wird eine ‘IllegalArgumentException‘-Ausnahme ausgelöst. Diese Ausnahme wird mit der Nachricht “`String-Array-Parameter kann keinen Nullwert annehmen!`” geworfen, was bedeutet, dass der String-Array-Parameter nicht null sein darf.

```
148     /**
149      * Die Methode bekommt ein zum Teil belegtes Hotel übergeben und gibt
150      * ein gültig voll belegtes Hotel zurück, wenn dies möglich ist. Die
151      * Details, was "gültig" bedeutet, stehen auf dem Übungsblatt.
152      *
153      * @param hotel Das zum Teil belegte Hotel. Es besteht aus den Zeichen
154      *              ' ', 'O' und 'X'. Alle Zeilen müssen dieselbe Länge
155      *              haben. Weder der Parameter noch eine seiner Zeilen
156      *              dürfen null sein. Wenn von diesen Vorgaben, abgewichen
157      *              wird, ist das Verhalten undefiniert.
158      * @return Ein gültig voll belegtes Hotel oder null, wenn es keine
159      * gültige Belegung gibt.
160      */
161     public static String[] solve(final String[] hotel) {
162 
163         //GIBT EINE FEHLERMELDUNG AUS, WENN DAS FELD LEER IST ODER WENN DIE GRÖSSE UND
                DIE ANZAHL DER ELEMENTE KEIN VIELFACHES VON ZWEI SIND
164         if (hotel == null)
165             throw new IllegalArgumentException("String-Array-Parameter kann keinen
                    Nullwert annehmen!");
```

Dann wird die Länge des hotel-Arrays überprüft, und wenn sie 0 beträgt, was bedeutet, dass es keine Elemente enthält, wird eine ‘IllegalArgumentException‘-Ausnahme ausgelöst. - Diese Ausnahme wird mit der Nachricht “`String-Array-Parameter darf nicht leer sein!`” geworfen, was bedeutet, dass der String-Array-Parameter nicht leer sein darf.

```
167         if (hotel.length == 0)
168             throw new IllegalArgumentException("String-Array-Parameter darf nicht leer
                    sein!");
```

Schließlich wird die Größe des Arrays und die Anzahl der Elemente überprüft. Wenn die Größe des Arrays und die Anzahl der Elemente durch zwei nicht ohne Rest teilbar kann, dann (was bedeutet, dass es keine Möglichkeit im Array keine gleiche Anzahl von X- und O-Zeichen gibt), wird eine ‘IllegalArgumentException‘-Ausnahme ausgelöst. - Diese Ausnahme wird mit einer Nachricht geworfen, die besagt, dass die Größe des Arrays und die Anzahl der Elemente nicht um Vielfache von 2 unterschiedlich sein dürfen, und dass X- und O-Zeichen horizontal und vertikal in gleicher Anzahl vorhanden sein müssen

```
170         if (hotel.length % 2 != 0 || hotel[0].length() % 2 != 0) {
171             String throwMessage = "\n" + "Aufgrund von Kontrollvorschriften dürfen sich
                        die Größe des Feldes und die Anzahl der Elemente nicht um Vielfache
                        von 2 unterscheiden!";
172             throwMessage += "\n" + "X- und O-Zeichen müssen horizontal und vertikal in
                        gleicher Anzahl vorhanden sein.";
173 
174             throw new IllegalArgumentException(throwMessage);
175         }
```

Am Ende der Abfolge werden nacheinander die Werte X und O gewählt (die Reihenfolge ist egal) und das Ergebnis mit check überprüft

-INVALID: Die aktuelle Abfolge ist ungültig, daher muss für das letzte Element der Abfolge der nächste Wert ausprobiert werden.
- VALID: Die aktuelle Belegung ist gültig, also kann ein weiteres Element an die Abfolge angehängt werden, das durchprobiert wird.
- SOLVED: Das Rätsel ist gelöst und wird als Ergebnis zurückgeliefert.

Immer wenn das letzte Element der Abfolge bereits erfolglos X und O durchlaufen hat, muss Backtracking angewendet werden, d. h. es wird wieder entfernt und das Element davor "weitergezählt". Entsteht beim Backtracking die leere Abfolge, gibt es keine Lösung. Dann gibt solve null statt einer Lösung zurück.

```
177         StringBuilder newData = new StringBuilder();
178 
179         while (true) {
180             final String[] copyOfHotel = hotel.clone();
181 
182             fill(copyOfHotel, newData.toString());
183 
184             Result result = check(copyOfHotel);
185             if (result == Result.VALID) {
186                 newData.append("O");
187             } else if (result == Result.INVALID) {
188                 final int position = newData.lastIndexOf("O");
189                 if (position == -1) {
190                     return null;
191                 }
192                 newData.delete(position, newData.length());
193                 newData.append("X");
194             } else if (result == Result.SOLVED) {
195                 return copyOfHotel;
196             }
197         }
198     }
199 }
```

InsectHotelTest.java

Die InsectHotelTest Klasse wurde von **Altug Uyanik** verfasst.

Die folgenden Tests überprüfen, ob die verschiedenen Methoden der Klasse InsectHotel korrekt funktionieren.

```java
package de.uni_bremen.pi2;

// INVALID, VALID und SOLVED können direkt verwendet werden.
import static de.uni_bremen.pi2.InsectHotel.Result.*;
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;
/**
 *Author Altug Uyanik
 */
public class InsectHotelTest
{
    /**
     * Ein komplexer Test aus dem ursprünglichen Puzzle.
     * Ist standardmäßig deaktiviert.
     */
    @Test
    public void testComplex()
    {
        final String[] puzzle = {
                "X       ",
                " OO     ",
                "   X   O",
                "O       ",
                "  XO  X ",
                "X X     ",
                "     X  ",
                "     X  "
        };
        final String[] solution = {
                "XXOOXOOX",
                "XOOXOOXX",
                "OOXXOXXO",
                "OXOOXXOX",
                "XOXOXOXO",
                "XOXXOOXO",
                "OXOXOXOX",
                "OXXOXXOO"
        };
        assertArrayEquals(solution, InsectHotel.solve(puzzle));
    }
```

Aufgabe 1.1 Ausfüllen

Diese beiden Testmethoden prüfen, wie die `fill`-Methode in verschiedenen Situationen funktioniert. Die Methode `testFill_EmptyEntries` testet den Fall, wenn das Hotel bereits belegt ist und die eingefügten Einträge leer sind.Das Hotel-Array ist auf eine bestimmte Weise definiert, und wenn die `fill`-Methode mit einem leeren Array aufgerufen wird, sollte das Hotel-Array unverändert bleiben. In diesem Fall sollte die Methode das ursprüngliche Hotel-Array zurückgeben.

```java
    /**
     * Test für die fill-Methode, wenn die Hoteleinträge leer sind.
     */
    @Test
    public void testFill_EmptyEntries() {
        final String[] hotel = {"XO  ", "OX X", "X O ", "O X "};
        final String[] result = InsectHotel.fill(hotel, "");
        assertArrayEquals(hotel, result);
    }
```

Die Methode `testFill_FullEntries` hingegen testet den Fall, wenn das Hotel bereits belegt ist und die eingefügten Einträge vollständig belegt sind. Das gleiche Hotel-Array wird verwendet, und wenn die `fill`-Methode mit den angegebenen Einträgen aufgerufen wird, sollten die leeren Zellen mit diesen neuen Einträgen gefüllt werden. Beide Tests überprüfen, ob das von der Methode zurückgegebene Ergebnis mit dem ursprünglichen Hotel-Array übereinstimmt.

```
52      /**
53       * Test für die fill-Methode, wenn die Hoteleinträge voll sind.
54       */
55      @Test
56      public void testFill_FullEntries() {
57          final String[] hotel = {"XO  ", "OX X", "X O ", "O X "};
58          final String[] result = InsectHotel.fill(hotel, "XOOOXXO");
59          assertArrayEquals(hotel, result);
60      }
```

Aufgabe 1.2 Prüfen

testCheckValid

Beschreibung: Dieser Test überprüft, ob die Methode `check` eine gültige Hotelkonfiguration korrekt erkennt.

Szenario: Ein Hotelarray ist wie folgt definiert:

```
{"XO  ", "OX X", "X OO", "OOX "}
```

Erwartetes Verhalten: Die Methode `check` sollte dieses Hotelarray als gültig erkennen.

Überprüfung: Der Ausdruck `assertEquals(InsectHotel.Result.VALID, InsectHotel.check(hotel))` überprüft, ob das von der Methode `check` zurückgegebene Ergebnis dem erwarteten Ergebnis entspricht.

```
63      /**
64       * Test für die check-Methode zur Validierung einer gültigen Hotelkonfiguration.
65       */
66      @Test
67      public void testCheckValid() {
68          String[] hotel = {"XO  ", "OX X", "X OO", "OOX "};
69          assertEquals(InsectHotel.Result.VALID, InsectHotel.check(hotel));
70      }
```

testCheckInvalid

Beschreibung: Dieser Test überprüft, ob die Methode `check` eine ungültige Hotelkonfiguration korrekt erkennt.

Szenario: Ein Hotelarray ist wie folgt definiert:

```
{"XXXO", "XOXO  ", "OXOX", "XXOO"}
```

Erwartetes Verhalten: Die Methode `check` sollte dieses Hotelarray als ungültig erkennen.

Überprüfung: Der Ausdruck `assertEquals(InsectHotel.Result.INVALID, InsectHotel.check(hotel))` überprüft, ob das von der Methode `check` zurückgegebene Ergebnis dem erwarteten Ergebnis entspricht.

```
72      /**
73       * Test auf eine check-Methode zur Erkennung einer ungültigen Hotelkonfiguration.
74       */
75      @Test
76      public void testCheckInvalid() {
77          String[] hotel = {"XXXO", "XOXO  ", "OXOX", "XXOO"};
78          assertEquals(InsectHotel.Result.INVALID, InsectHotel.check(hotel));
79      }
```

testCheckSolved

Beschreibung: Dieser Test überprüft, ob die Methode `check` eine gelöste Hotelkonfiguration korrekt erkennt.

Szenario: Ein Hotelarray ist wie folgt definiert:

```
{"XOXO", "OXOX", "XXOO", "OOXX"}
```

Erwartetes Verhalten: Die Methode `check` sollte dieses Hotelarray als gelöst erkennen.

Überprüfung: Der Ausdruck `assertEquals(InsectHotel.Result.SOLVED, InsectHotel.check(hotel))` überprüft, ob das von der Methode `check` zurückgegebene Ergebnis dem erwarteten Ergebnis entspricht.

```
82      /**
83       * Test auf eine check-Methode zur Erkennung einer gelösten Hotelkonfiguration.
84       */
85      @Test
86      public void testCheckSolved() {
87          String[] hotel = {"XOXO", "OXOX", "XXOO", "OOXX"};
88          assertEquals(InsectHotel.Result.SOLVED, InsectHotel.check(hotel));
89      }
```

testCheck_AllX

Beschreibung: Dieser Test überprüft, ob die Methode `check` korrekt erkennt, dass das Hotel komplett mit 'X' gefüllt ist.

Szenario: Ein Hotelarray ist wie folgt definiert:

```
{"XXXX", "XXXX", "XXXX", "XXXX"}
```

Erwartetes Verhalten: Die Methode `check` sollte dieses Hotelarray als ungültig erkennen.

Überprüfung: Der Ausdruck `assertEquals(InsectHotel.Result.INVALID, InsectHotel.check(hotel))` überprüft, ob das von der Methode `check` zurückgegebene Ergebnis dem erwarteten Ergebnis entspricht.

```
91      /**
92       * Prüfung der check-Methode, wenn das Hotel vollständig mit    X    gefüllt ist.
93       */
94      @Test
95      public void testCheck_AllX() {
96          final String[] hotel = {"XXXX", "XXXX", "XXXX", "XXXX"};
97          assertEquals(INVALID, InsectHotel.check(hotel));
98      }
```

testCheck_ValidHotel

Beschreibung: Dieser Test überprüft, ob die Methode `check` eine andere gültige Hotelkonfiguration korrekt erkennt.

Szenario: Ein Hotelarray ist wie folgt definiert:

```
{"XO  ", "OX X", "X OO", "OOX "}
```

Erwartetes Verhalten: Die Methode `check` sollte dieses Hotelarray als gültig erkennen.

Überprüfung: Der Ausdruck `assertEquals(InsectHotel.Result.VALID, InsectHotel.check(hotel))` überprüft, ob das von der Methode `check` zurückgegebene Ergebnis dem erwarteten Ergebnis entspricht.

```
100     /**
101      * Doppelter Test für die check-Methode mit einer gültigen Hotelkonfiguration.
102      */
```

```
103     @Test
104     public void testCheck_ValidHotel() {
105         final String[] hotel = {"XO  ", "OX X", "X OO", "OOX "};
106         assertEquals(InsectHotel.Result.VALID, InsectHotel.check(hotel));
107     }
```

testCheck_InvalidPattern

Beschreibung: Dieser Test überprüft, ob die Methode `check` korrekt erkennt, dass das Hotel ein ungültiges Muster enthält.

Szenario: Ein Hotelarray ist wie folgt definiert:

```
{"XXXO", "OOOX", "XOXX", "OXOO"}
```

Erwartetes Verhalten: Die Methode `check` sollte dieses Hotelarray als ungültig erkennen.

Überprüfung: Der Ausdruck `assertEquals(InsectHotel.Result.INVALID, InsectHotel.check(hotel))` überprüft, ob das von der Methode `check` zurückgegebene Ergebnis dem erwarteten Ergebnis entspricht.

```
107
108     /**
109      * Test für die check-Methode, wenn das Hotel ein ungültiges Muster hat.
110      */
111     @Test
112     public void testCheck_InvalidPattern() {
113         final String[] hotel = {"XXXO", "OOOX", "XOXX", "OXOO"};
114         assertEquals(INVALID, InsectHotel.check(hotel));
115     }
```

testCheck_NullHotel

Beschreibung: Dieser Test überprüft, ob die Methode `check` eine angemessene Ausnahme wirft, wenn das Hotelarray null ist.

Erwartetes Verhalten: Es wird erwartet, dass eine `NullPointerException` geworfen wird.

Überprüfung: Der Ausdruck `assertThrowsExactly(NullPointerException.class, () -> InsectHotel.check(null))` überprüft, ob die Methode `check` die erwartete Ausnahme angemessen wirft.

```
116     /**
117      * Test für die check-Methode, wenn das Hotel null ist.
118      */
119     @Test
120     public void testCheck_NullHotel() {
121         assertThrowsExactly(NullPointerException.class, () -> InsectHotel.check(null));
122     }
```

testCheck_SolvedHotel

Beschreibung: Dieser Test überprüft, ob die Methode `check` eine von der `solve`-Methode zurückgegebene gelöste Hotelkonfiguration korrekt erkennt.

Szenario: Ein Hotelarray ist wie folgt definiert und seine gelöste Version wird aus der `solve`-Methode erhalten.

Erwartetes Verhalten: Die Methode `check` sollte diese gelöste Hotelkonfiguration als `SOLVED` erkennen.

Überprüfung: Der Ausdruck `assertEquals(InsectHotel.Result.SOLVED, InsectHotel.check(solvedHotel))` überprüft, ob das von der Methode `check` zurückgegebene Ergebnis dem erwarteten Ergebnis entspricht.

```
126     /**
127      * Test für die check-Methode zur Überprüfung einer gelösten Hotelkonfiguration, die
             von der solve-Methode zurückgegeben wird.
```

```
128      */
129     @Test
130     public void testCheck_SolvedHotel() {
131         final String[] hotel = {"XO  ", "OX X", "X OO", "OOX "};
132         final String[] solvedHotel = InsectHotel.solve(hotel);
133         assertEquals(SOLVED, InsectHotel.check(solvedHotel));
134     }
```

testCheck_EmptyHotel

Beschreibung: Dieser Test überprüft, ob die Methode `check` eine angemessene Ausnahme wirft, wenn das Hotelarray leer ist.

Erwartetes Verhalten: Es wird erwartet, dass eine `IllegalArgumentException` geworfen wird.

Überprüfung: Der Ausdruck `assertThrowsExactly(IllegalArgumentException.class, () -> InsectHotel.solve(hotel))` überprüft, ob die Methode `check` die erwartete Ausnahme angemessen wirft.

```
136     /**
137      * Test für die check-Methode, wenn das Hotelarray leer ist.
138      */
139     @Test
140     public void testCheck_EmptyHotel() {
141         final String[] hotel = {};
142         assertThrowsExactly(IllegalArgumentException.class, () -> InsectHotel.solve(hotel
                ));
143     }
```

Aufgabe 1.3 Lösen

testSolve_InvalidHotel

- Dieser Test überprüft, ob die ‘solve‘-Methode korrekt ‘null‘ zurückgibt, wenn sie eine ungültige Hotelkonfiguration erhält. Zum Beispiel, wenn eine ungültige Konfiguration wie

```
"XXX ", "OOO ", "X XX", "OOO "
```

übergeben wird, sollte die Methode ‘null‘ zurückgeben.

```
145     /**
146      * Test auf solve-Methode, wenn die Hotelkonfiguration ungültig ist.
147      */
148     @Test
149     public void testSolve_InvalidHotel() {
150         final String[] hotel = {"XXX ", "OOO ", "X XX", "OOO "};
151         assertNull(InsectHotel.solve(hotel));
152     }
```

testSolve_ValidHotel

- Dieser Test überprüft, ob die ‘solve‘-Methode für eine gültige Hotelkonfiguration die richtige Lösung generiert. Zum Beispiel wird für eine gültige Konfiguration wie

```
"XO  ", "OX X", "X OO", "OOX "
```

die erwartete Lösung als

```
"XOXO", "OXOX", "XXOO", "OOXX"
```

angegeben.

```java
    /**
     * Test für die solve-Methode zur Lösung einer gültigen Hotelkonfiguration.
     */
    @Test
    public void testSolve_ValidHotel() {
        final String[] hotel = {"XO  ", "OX X", "X OO", "OOX "};
        final String[] solution = {"XOXO", "OXOX", "XXOO", "OOXX"};
        assertArrayEquals(solution, InsectHotel.solve(hotel));
    }
```

testSolve_NullHotel

- Dieser Test überprüft, ob die ‘solve‘-Methode eine ‘IllegalArgumentException‘ wirft, wenn sie mit ‘null‘ als Argument aufgerufen wird. Das bedeutet, dass die Methode eine angemessene Ausnahme auslösen sollte, wenn sie mit ‘null‘ aufgerufen wird.

```java
    /**
     * Test für die Methode solve, wenn das Hotel null ist.
     */
    @Test
    public void testSolve_NullHotel() {
        assertThrowsExactly(IllegalArgumentException.class, () -> InsectHotel.solve(null)
                );
    }
```

testSolve_EmptyHotel

- Dieser Test überprüft, ob die ‘solve‘-Methode eine ‘IllegalArgumentException‘ wirft, wenn sie mit einer leeren Hotelkonfiguration aufgerufen wird. Eine leere Konfiguration bedeutet, dass keine Zimmer angegeben sind oder keine Insekten in einem Zimmer platziert sind.

```java
    /**
     * Test für die Methode solve, wenn das Hotel-Array leer ist.
     */
    @Test
    public void testSolve_EmptyHotel() {
        final String[] hotel = {" "};
        assertThrowsExactly(IllegalArgumentException.class, () -> InsectHotel.solve(hotel
                ));
    }
```

testSolve_ImpossibleHotel

- Dieser Test überprüft, ob die ‘solve‘-Methode korrekt ‘null‘ zurückgibt, wenn sie eine unlösbare Hotelkonfiguration erhält. Zum Beispiel sollte die Methode für eine unlösbare Konfiguration wie

```
"XXOO", "XXOO", "XXOO", "XXOO"
```

‘null‘ zurückgeben.

```java
    /**
     * Test für die solve-Methode, wenn die Hotelkonfiguration unmöglich zu lösen ist.
     */
    @Test
    public void testSolve_ImpossibleHotel() {
        final String[] hotel = {"XXOO", "XXOO", "XXOO", "XXOO"};
        assertNull(InsectHotel.solve(hotel));
    }
}
```