Dokumentation Projektarbeit Carcasonne

Leopold Keller, Laurenz Kammeyer, Frederick Wichert, Jonas Milkovits

Inhaltsverzeichnis

1 Dokumentation - 6.1.4(b)

2 Dokumentation - 6.1.5

3 Dokumentation - 6.3.1

Die graphische Oberfläche wurde um folgende Elemente erweitert:

- **Tabelle**
  - **Ziel der Spaltengestaltung:**
    1. Möglichst übersichtliche Gestaltung der Spalten
    2. kurze, aussagekräftige Namen
    3. Informationen so übersichtlich wie möglich → wenige Spalten
  - Die Tabelle wurde mit folgendem Code erstellt:

```
1     private JTable scoreTable;
2     List<ScoreEntry> list = resources.getScoreEntries();
3     Object[] column = { "Date", "Name", "Score" };
4     Object[][] data = new Object[list.size()][3];
5     scoreTable = new JTable(data, column);
```

  - Das zweidimensionale Array `data` wurde mit folgendem Code befüllt:

```
1    for (int i = 0; i < list.size(); i++) {
2     if (list.get(i) != null) {
3      data[i][0] = list.get(i).getDate();
4      data[i][1] = list.get(i).getName();
5      data[i][2] = list.get(i).getScore();
6     }
7    }
```

  - **Problem:** Beim Standardmodell editierbare Zellen

    ⇒ **Lösung:** Eigenes Table-Modell

    Das Überschreiben der Methode `isCellEditable(int row, int column)` erlaubt es uns das Editieren der Zellen zu verhindern. Das Überschreiben der zweiten Methode benötigen wir später für das Sortieren der Spalten.

```
1    DefaultTableModel tableModel = new DefaultTableModel(data, column) {
2
3     @Override
4     public boolean isCellEditable(int row, int column) {
5      // all cells false
6      return false;
7     }
8
9     public Class<?> getColumnClass(int columnIndex) {
10     return getValueAt(0, columnIndex).getClass();
11    }
12
13   };
14
15   scoreTable.setModel(tableModel);
```

  - **Ergebnis:**

    Wir haben eine Tabelle mit drei Spalten (Datum, Name, Punkte) erstellt. Das Datumsformat ist absichtlich kürzer gehalten, da unserer Meinung nach der Tag des Spiels ausreichend ist.

- **Scrollbar**
  - Eine Scrollbar wird benötigt, falls zuviele Daten vorhanden sind
  - Die Scrollbar wurde mit folgendem Code erstellt:

```
1    private JScrollPane scrollPane;
2    scrollPane = new JScrollPane(scoreTable);
3    add(scrollPane);
```

  - Dies verknüpft die Tabelle mit der Scrollbar

- **Sortierbarkeit der Spalten**

  Die Sortierbarkeit der Spalten wurde uns hier zwar nicht vorgegeben, war für uns aber ein wichtiges Feature bei einer Highscore-Tabelle. Diese dient auch der einfacheren Verwendbarkeit für den Endnutzer. Alle Spalten sind nach ihren jeweiligen Werten sortierbar.

  - Dies wurde mit folgendem Code erreicht:

```
1    TableRowSorter<TableModel> sorter = new TableRowSorter<TableModel>(scoreTable.
         getModel());
2    scoreTable.setRowSorter(sorter);
```

  Die Vorarbeit, die hierfür benötigt wurde, ist weiter oben in der Erstellung von `DefaultTableModel` zu finden. Dies erlaubt es uns, auch das Datum sowie die Punktzahl zu sortieren. Ohne diese Vorarbeit könnte man das Ganze nur korrekt nach Namen sortieren, da der Autosorter den Wert des Strings vergleicht, was bei dem Datum und der Punktzahl zu falschen Sortierungen führt.

4 Dokumentation - 6.3.2

5 Dokumentation - 6.3.3

6 Dokumentation - 6.3.4

Der Computergegner besteht aus zwei Methoden:

- `draw(GamePlay gp, Tile tile)`

  Als ersten Schritt suchen wir uns alle möglichen Platziermöglichkeiten.
  Dies geschieht indem wir die Liste der platzierten Tiles des Boards durchlaufen und bei jedem Tile alle Positionen (nördlich, südlich, westlich, östlich) überprüfen und diese, falls das zu platzierende Tile anlegbar ist, abspeichern.
  Dieses Abspeichern geschieht in einer Liste einer selbst erstellten Klasse namens PointRotation, die Koordinaten und die Rotation der Position abspeichern. Danach lassen wir uns eine zufällige Zahl (zwischen 0 und der Länge Liste - 1) ausgeben und verwenden diese zufällige Position dann um an dieser Stelle dann das zu platzierende Tile zu platzieren.

  – Suche der möglichen Platzierungsmöglichkeiten:

```
1   List<Tile> tiles = gc.getGameBoard().getTiles();
2    List<PointRotation> possibleLocations = new ArrayList<PointRotation>();
3
4    for (Tile t : tiles) {
5     // check top
6     for (int r = 0; r < 4; r++) {
7      if (gc.getGameBoard().isTileAllowed(topTile, t.x, t.y - 1) && !
8      gc.getGameBoard().isTileAtPosition(t.x, t.y - 1)) {
9       possibleLocations.add(new PointRotation(t.x, t.y - 1, topTile.getRotation()));
10     }
11     topTile.rotateRight();
12    }
13
14    // check right
15    for (int r = 0; r < 4; r++) {
16     if (gc.getGameBoard().isTileAllowed(topTile, t.x + 1, t.y) && !
17
18     gc.getGameBoard().isTileAtPosition(t.x + 1, t.y)) {
19      possibleLocations.add(new PointRotation(t.x + 1, t.y, topTile.getRotation()));
20     }
21     topTile.rotateRight();
22    }
23
24    // check left
25    for (int r = 0; r < 4; r++) {
26     if (gc.getGameBoard().isTileAllowed(topTile, t.x - 1, t.y) && !
27     gc.getGameBoard().isTileAtPosition(t.x - 1, t.y)) {
28      possibleLocations.add(new PointRotation(t.x - 1, t.y, topTile.getRotation()));
29     }
30     topTile.rotateRight();
31    }
32
33    // check bottom
34    for (int r = 0; r < 4; r++) {
35     if (gc.getGameBoard().isTileAllowed(topTile, t.x, t.y + 1) && !
36     gc.getGameBoard().isTileAtPosition(t.x, t.y + 1)) {
37      possibleLocations.add(new PointRotation(t.x, t.y + 1, topTile.getRotation()));
38     }
39     topTile.rotateRight();
40    }
41   }
```

  – Suche nach zufälliger Zahl:

```
1   Random randomGen = new Random();
2   int randomNum = randomGen.nextInt(possibleLocations.size());
3   PointRotation randomLocation = possibleLocations.get(randomNum);
```

– Rotation der zu platzierenden Tile und Platzieren dieser:

```
1   // Rotate tile to be placed
2   while (topTile.getRotation() != randomLocation.getRotation()) {
3    topTile.rotateRight();
4   }
5
6   // Add rotated tile to GameBoard
7   gc.getGameBoard().newTile(topTile, randomLocation.getX(), randomLocation.getY());
```

- `placeMeeple(GamePlay gp)`

  Das Platzieren des Meeples läuft folgendermaßen ab. Wir speichern mithilfe der Methode `getMeepleSpots()` alle möglichen MeepleSpots des zuletzt platzierten Tiles ab und nutzen diese um uns eine ArrayList des Types Position zu erstellen. In dieser werden alle Positionen gespeichert, an denen ein Meeple platzierbar ist. Wir lassen uns dann wieder eine zufällige Zahl ausgeben (basierend auf der Länge der ArrayListe) und platzieren den Meeple an dieser Stelle, falls die Anzahl der Meeple größer als 0 ist. Falls es keinen verfügbaren MeepleSpot gibt, wird die Methode `nextRound()` aufgerufen, die die momentane Runde beendet.

  – Überprüfung, ob MeepleSpot vorhanden ist:

```
1   if (meepleSpots == null) {
2    gp.nextRound();
3    return;
4   }
```

  – Abspeichern aller möglichen MeepleSpots:

```
1   // boolean array of length 9, true where meeple can be placed on current tile
2   boolean[] meepleSpots = gc.getGameBoard().getMeepleSpots();
3
4   // array with all positions
5   Position[] positions = Position.getAllPosition();
6
7   // Puts all positions with a valid meeple spot into an ArrayList
8   ArrayList<Position> possibleMeepleSpots = new ArrayList<Position>();
9
10  for (int i = 0; i < positions.length; i++) {
11   if (meepleSpots[i] == true) {
12    possibleMeepleSpots.add(positions[i]);
13   }
14  }
```

  – Zufälliger MeepleSpot und Platzieren des Meeples:

```
1   // Getting random meeple spot
2   Random randomGen = new Random();
3   int randomNum = randomGen.nextInt(possibleMeepleSpots.size());
4
5   Position randomMeepleSpot = possibleMeepleSpots.get(randomNum);
6
7   if(meeples > 0) {
8    gp.placeMeeple(randomMeepleSpot);
9   }
10  else {
11   gp.nextRound();
12  }
```