Prof. Heike Wehrheim
und Mitarbeiter

Paderborn, den 7. Januar 2008

Musterlösung zur Weihnachtsübung
Grundlagen der Programmierung 1
WS 2007/08

AUFGABE 1:

In der Weihnachtsaufgabe sollten die Rümpfe der kursiv dargestellten Methoden der vorgegebenen Klassen `GameOfLife` und `GameOfLifeField` aus Abbildung 1 implementiert werden.

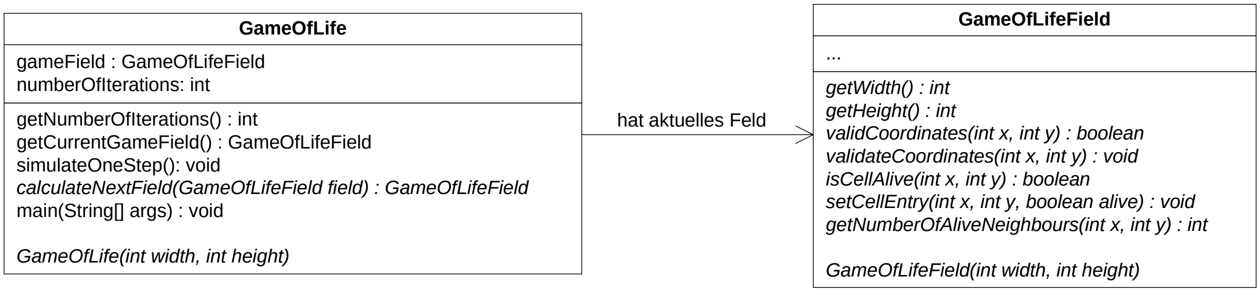
GameOfLife
gameField : GameOfLifeField
numberOfIterations: int
getNumberOfIterations() : int
getCurrentGameField() : GameOfLifeField
simulateOneStep(): void
calculateNextField(GameOfLifeField field) : GameOfLifeField
main(String[] args) : void
GameOfLife(int width, int height)
hat aktuelles Feld
GameOfLifeField
...
getWidth() : int
getHeight() : int
validCoordinates(int x, int y) : boolean
validateCoordinates(int x, int y) : void
isCellAlive(int x, int y) : boolean
setCellEntry(int x, int y, boolean alive) : void
getNumberOfAliveNeighbours(int x, int y) : int
GameOfLifeField(int width, int height)

Abbildung 1: Zu erweiternde Klassen

Die Klasse `GameOfLifeField` repräsentiert das Spielfeld und soll die Zustände aller Zellen speichern, also merken, ob eine Zelle ein lebendiges Lebewesen enthält oder nicht. Das kann mit Hilfe eines zweidimensionalen Arrays mit `boolean`-Einträgen gelöst werden. Eine mögliche Implementierung der Klasse `GameOfLifeField` könnte also wie folgt aussehen.

```java
/**
 * This class represents the field of a "Game of Life" game instance,
 * i.e. a 2-dimensional grid of cells, where one cell can be empty
 * or contain a living creature.
 *
 * @author Dietrich Travkin
 */
public class GameOfLifeField
{
    private boolean[][] matrix;

    /**
     * Creates a new field with the given width and height,
     * i.e. a grid field with 'width' cells in each row and
     * 'height' cells in each column.
     *
     * @param width number of cells in each row (number of columns)
     * @param height number of cells in each column (number of rows)
     */
    public GameOfLifeField(int width, int height)
    {
        this.matrix = new boolean[height][width];
    }

    /**
     * Returns the width of the field (number of cells in a row).
     *
     * @return the width of the field (number of cells in a row).
     */
    public int getWidth()
    {
        return (this.matrix.length > 0 ? this.matrix[0].length : 0);
    }

    /**
     * Returns the height of the field (number of cells in a column).
     *
     * @return the height of the field (number of cells in a column).
     */
    public int getHeight()
    {
        return this.matrix.length;
    }

    /**
     * Determines whether the given coordinates are within the
     * game field bounds.
     *
     * @param x the x coordinate (column)
     * @param y the y coordinate (row)
     * @return <code>true</code> if the coordinates are within
     *         field bounds, <code>false</code> otherwise.
     */
    public boolean validCoordinates(int x, int y)
    {
        return (x >= 0 && x < getWidth() && y >= 0 && y < getHeight());
    }

    /**
     * Checks whether the given coordinates are whithin the
     * game field bounds and throws an
     * <code>IllegalArgumentException</code>, if they are not.
     *
     * @param x the x coordinate (column)
     * @param y the y coordinate (row)
     */
    private void validateCoordinates(int x, int y)
    {
        if (!this.validCoordinates(x, y))
        {
            throw new IllegalArgumentException("Coordinates ("
                    + x + ", " + y + ") are out of bounds.");
        }
    }

    /**
     * Returns true, if the cell with coordinate (x,y) is alive,
     * false otherwise (especially, if the field does not exist).
     *
     * Lowest valid coordinate value is 0,
     * highest valid x value is width - 1,
     * highest valid y value is height - 1.
     *
     * @param x the x coordinate of the cell (column)
     * @param y the y coordinate of the cell (row)
     * @return true, if the cell is alive, false otherwise.
     */
    public boolean isCellAlive(int x, int y)
    {
        this.validateCoordinates(x, y);

        return matrix[y][x];
    }

    /**
     * Sets the alive value of the cell with the given coordinate.
     *
     * Lowest valid coordinate value is 0,
     * highest valid x value is width - 1,
     * highest valid y value is height - 1.
     *
     * @param x the x coordinate of the cell (column)
     * @param y the y coordinate of the cell (row)
     * @param alive the alive value
     */
    public void setCellEntry(int x, int y, boolean alive)
    {
        this.validateCoordinates(x, y);

        matrix[y][x] = alive;
    }

    /**
     * Determines the number of alive neighbours of the cell
     * with coordinates (x, y).
     *
     * Lowest valid coordinate value is 0,
     * highest valid x value is width - 1,
     * highest valid y value is height - 1.
     *
     * @param x the x coordinate of the cell (column)
     * @param y the y coordinate of the cell (row)
     * @return number of alive neighbour cells
     */
    public int getNumberOfAliveNeighbours(int x, int y)
    {
        this.validateCoordinates(x, y);

        // Run through all neighbours.
        // The cell given by x and y is in the center,
        // the surrounding cells have to be checked.
        //
        // |1|2|3|
        // |4| |5|
        // |6|7|8|

        int result = 0;

        for (int row = y - 1; row <= y + 1; row++)
        {
            for (int column = x - 1; column <= x + 1; column++)
            {
                if ( (row != y || column != x) // not (x,y) coordinates
                        && this.validCoordinates(column, row) // whithin field bounds
                        && matrix[row][column] ) // cell is alive
                {
                    result++;
                }
            }
        }

        return result;
    }
}
```

Eine weitere Aufgabe war es, die Klasse `GameOfLife` zu erweitern. Dazu muss im Konstruktor ein `GameOfLifeField`-Objekt erzeugt werden (siehe Zeile 19) und der Rumpf der Methode `calculateNextField` implementiert werden (siehe Zeilen 59 bis 98).

```java
1  /**
2   * This class represents instances of "Game of Life" simulations/games.
3   *
4   * @author Dietrich Travkin
5   */
6  public class GameOfLife
7  {
8      private GameOfLifeField gameField;
```

```
   private int numberOfIterations = 0;

   /**
    * Creates a new game instance with the given field width and height.
    *
    * @param width the width of the game field
    * @param height the height of the game field
    */
   public GameOfLife(int width, int height)
   {
      this.gameField = new GameOfLifeField(width, height);
   }

   /**
    * Returns the number of simulation iterations already run or
    * the number of the current generation.
    *
    * @return the number of game iterations (population growth steps).
    */
   public int getNumberOfIterations()
   {
      return numberOfIterations;
   }

   /**
    * Returns the game field of the current simulation step
    * (current population).
    *
    * @return the game field of the current simulation step
    */
   public GameOfLifeField getCurrentGameField()
   {
      if (this.gameField == null)
      {
         throw new IllegalStateException(”Current game field is null!”);
      }
      return this.gameField;
   }

   /**
    * Run one simulation step, i.e. determine the population
    * of the next generation and change the game field
    * accordingly.
    */
   public void simulateOneStep()
   {
      this.gameField = this.calculateNextField(this.gameField);
      this.numberOfIterations++;
   }

   /**
    * Given a current population (in a given GameOfLifeField object)
    * this method determines the next generation's population and
    * returns it in a new GameOfLifeField object.
    *
    * @param field the current population
    * @return the next generation's population
    */
   private GameOfLifeField calculateNextField (GameOfLifeField field)
   {
      GameOfLifeField nextField = new GameOfLifeField(
              field.getWidth(), field.getHeight());

      for (int x = 0; x < field.getWidth(); x++)
      {
          for (int y = 0; y < field.getHeight(); y++)
         {
              // initialize cell of new field with cell entry of old field
              nextField.setCellEntry(x, y, field.isCellAlive(x, y));

              // determine the new cell value for the new field
              int numberOfNeighbors = field.getNumberOfAliveNeighbours(x, y);
              if (numberOfNeighbors < 2 || numberOfNeighbors > 3)
              {
                  // loneliness or crowding, cell entry dies (if existent)
                  nextField.setCellEntry(x, y, false);
              }
              else if (!field.isCellAlive(x, y)) // if cell is empty
              {
                  if (numberOfNeighbors == 3)
                  {
                      // new cell member is born
                      nextField.setCellEntry(x, y, true);
                  }
              }
         }
      }

      return nextField;
   }

   /**
    * Create a graphical user interface for a ”Game of Life” instance.
    *
    * @param args console arguments (no arguments expected)
    */
```

```java
    public static void main(String[] args)
    {
        GameOfLifeWindow window = new GameOfLifeWindow();
        window.setSize(640, 480);
        window.setVisible(true);
    }
}
```