Introduction to Programming - Re-exam Project (Part II)

Simon Dradrach Jørgensen
Supervisor: Jan Baumbach
DM550 - Re-Exam

7. september 2017

Indhold

1 Specification

The given task is to print an un-solved *sudoku puzzle* to the user, which they're supposed to solve. This means an implementation of a *GUI* is required, to show the output to the user, involving buttons to call functions that can load and save the file, (as long as it's a *Sudoku puzzle layout*).

Sudoku puzzles are a kind of crossword puzzles with numbers where the following two conditions have to be met: [1]

1.1 Condition 1

In each row or column, the nine numbers have to be from the *set [1,2,3,4,5,6,7,8,9]* and they must all be different.

1.2 Condition 2

For each of the nine non-overlapping 3x3 blocks, the nine numbers have to be from the *set [1,2,3,4,5,6,7,8,9]* and they must all be different.

2 Design

The goal of the program, is to construct a *user interface*, using the *swing library* which is a part of the default *java* library. Therenext, the user should be able to edit the "empty slots"with integers, to try to solve the *sudoku puzzle*, and lastly the program needs to check, if the user is correct in their inputs.

A way to load any un-solved *sudoku puzzle* is a requirement, since re-using the same layout over and over again is no challenge, thus a way to load a *sudoku-template.txt* file can be implemented, and obviously a way to save that file when the user have completed the *sudoku puzzle*.

To make sure that the program can check up on users inputs, being either correct or wrong, therefore a way to check if there's any conflicting inputs with the already plotted integers are required, and also a way to forbid anything else than integers. To make sure that user understands when and where they made a mistake, a color can be used to show what's right and what's wrong, also where exactly in the *sudoku puzzle*.

3 Implementation

For the following task, a *Field* class is required, creating two *arrays*, each with the same constant, which are then multiplied to forge the *sudoku table*, which is used for inserting the integers.

A numerous boolean functions were required, to make sure that the program could check where the inputs would be located. Since there's a single *array*, formed by two other *arrays*, these check functions are:

$$f(y, v) = \forall x \in \ \mathbb{Z} \cap]0 : 9] : data_{(x,y)} \neq v \tag{1}$$

This function checks for all elements in a given row, and returns false, if the inserted integer is a duplicate of another integer already in the *array*. Same goes for checking columns, but with [i][j] being swapped.

$$f(x, y, v) = \forall x' \in \ \mathbb{Z} \cap]0 : 3], \forall y' \in \ \mathbb{Z} \cap]0 : 3] : data_{(\lfloor \frac{x}{\sqrt{9}} \rfloor * 3 + x', \lfloor \frac{y}{\sqrt{9}} \rfloor * 3 + y')} \neq v$$

Where v is defined as the *value*. In this function, the program checks what box one is currently occupying in.

```
1  public boolean tryValue(int val, int i, int j) {
2         if (!checkRow(val, i) || !checkCol(val, j) || !checkBox(val, i, j
               )) {
3             return false;
4         }
5
6         this.model[i][j] = val;
7         return true;
8
9     }
```

These functions returns false if the value cannot be placed in either the row, or the column.

```
1  public boolean isEmpty(int i, int j) {
2         return model[i][j] == 0;
```

Is a function that checks for an empty space in the array.

3.1 Graphical User Interface

The way the user inserts integers, works best by allowing them to write the given integer, in the cells. These cells however, needs to be changeable inside the *UI* itself, requiring the use of the *JTextField* class.However, the user needs a way to tell the program that they are done inserting integers, this was implemented via a *JButton*. For the *JButton* to actively change the *JTextField*, a *check* function was implemented, this function does the following:

- Warn the user if the integer conversion from the given string, failed.
- Tells the user which variables are constant/fixed, and will redo any changes done to these cells
- Tells the users whether or not an integer failed the *sudoku check*, with a color scheme. **Blue** if succeeded, and **Red** if failed.
- Finally, it inherits the files way of telling, that the variable of a cell is unknown. This means that all **0**'s will be replaced, with **X**'s

To *load* the wanted file, the *load* function, incapable of setting the size of the *sudoku table*, resets all the text fields, and the *sudoku field*. The function will load all the integers from the *sudoku* file, with the *sudokus* in-built *fromFile* function, where it will let the *check* function replace all **0**'s with **X**'s as per fitting the *sudoku* file *scheme*.

The *save* function is a bit more simple, it *prints* the *sudoku* file, and replaces all the "special character", e.g., $\{+, -, |\}$ from the *Field* class. The *save* function however, does not need to replace all **0**'s with **X**'s, since there won't be any when the *save* function is available to the user, given that they'd only be able to save when the *sudoku* have been completed.

4 Testing

To test the *sudoku puzzle*, a solver was created to test if the program would only spit out the correct result when checking, e.g.,

```
1  Field field = new Field();
2         field.fromFile(args[0]);
3         try {
4             solve(field, 0);
5         } catch (SolvedException e) {
6             System.out.println(field);
7         }
8
9  public static void solve(Field f, int i) throws SolvedException {
10        if (i >= Field.SIZE * Field.SIZE) {
11            throw new SolvedException();
12        }
13
14        if (!f.isEmpty(i % Field.SIZE, i / Field.SIZE)) {
15            solve(f, i + 1);
16            return;
17        }
```

```
18
19          for (int x = 1; x <= Field.SIZE; ++x) {
20              if (f.tryValue(x, i % Field.SIZE, i / Field.SIZE)) {
21                  solve(f, i + 1);
22              }
23
24          }
25          f.clear(i % Field.SIZE, i / Field.SIZE);
26      }
```

Given the program one of the *test.sudoku.txt* files, it was possible to see if the solver could find an answer, and if the program could distinguish between a right and wrong input from the user. With this information, one could see the finished result, e.g.,

		8	1			5		
9	6			8				
	5			3	6		9	8
		7		6	9		1	2
		6	8		7	3		
8	1		4	5		6		
4	2		6	7			8	
				4			6	3
		5			8	7		

3	7	8	1	9	4	5	2	6
9	6	4	2	8	5	1	3	7
1	5	2	7	3	6	4	9	8
5	4	7	3	6	9	8	1	2
2	9	6	8	1	7	3	5	4
8	1	3	4	5	2	6	7	9
4	2	1	6	7	3	9	8	5
7	8	9	5	4	1	2	6	3
6	3	5	9	2	8	7	4	1

Figur 1: Sudoku

One could then plot those integers given and see if the program accepted that as an answer. If it did, it'd prove to be correct.

This proved useful in the early stages of the program, due to the fact, that the user would get "false"everytime they'd click the *check button*, even if there were no integer with n-value in either *row* 'nor *column*. It was then easy to find the mistake and fix it.

5 Conclusion

With the useage of the *swing* library, one can create a well-functioning *GUI* for a *sudoku puzzle*. It could however easily have been improved, by sepparating the 9 boxes in the *sudoku table* by a grid of some type, to make it more clear that within those boxes, duplicates are not allowed. This could however been done, by using a *class* that extends the *swing* layout manager interface.

6 Appendix (source code)

6.1 Field

```
/**
 *
 * @author r41
 */
import java.io.*;
import java.util.*;

/**
 * Abstract Data Type for Sudoku playing field
 */
public class Field {

    public static final int SIZE = 9;

    public int model[][];

    public Field() {
//       make new array of size SIZExSIZE
        this.model = new int[SIZE][SIZE];
    }

    public void fromFile(String fileName) {
        try {
            Scanner sc = new Scanner(new File(fileName));
            fromScanner(sc, 0, 0);
        } catch (FileNotFoundException e) {
            // :-(
        }
    }

    private void fromScanner(Scanner sc, int i, int j) {
        if (i >= SIZE) {
            // all rows done!
        } else if (j >= SIZE) {
            // this row done - go to next!
            fromScanner(sc, i + 1, 0);
        } else {
            try {
                int val = Integer.parseInt(sc.next());
                this.model[i][j] = val;
            } catch (NumberFormatException e) {
                // skip this cell
            }
            fromScanner(sc, i, j + 1);
        }
    }

    @Override
    public String toString() {
```

```
50          StringBuilder res = new StringBuilder();
51          for (int i = 0; i < SIZE; i++) {
52              if (i % 3 == 0) {
53                  res.append("+-------+-------+-------+\n");
54              }
55              for (int j = 0; j < SIZE; j++) {
56                  if (j % 3 == 0) {
57                      res.append("| ");
58                  }
59                  int val = this.model[i][j];
60                  res.append(val + " ");
61              }
62              res.append("|\n");
63          }
64          res.append("+-------+-------+-------+");
65          return res.toString();
66      }
67
68      /**
69       * returns false if the value val cannot be placed at row i and
             column j.
70       * returns true and sets the cell to val otherwise.
71       */
72      public boolean tryValue(int val, int i, int j) {
73          if (!checkRow(val, i) || !checkCol(val, j) || !checkBox(val, i, j
                 )) {
74              return false;
75          }
76
77          this.model[i][j] = val;
78          return true;
79
80      }
81
82      /**
83       * checks if the cell at row i and column j is empty, i.e., whether
             it
84       * contains 0
85       */
86      public boolean isEmpty(int i, int j) {
87          return model[i][j] == 0;
88          //if value is 0, it'll return true
89      }
90
91      /**
92       * sets the cell at row i and column j to be empty, i.e., to be 0
93       */
94      public void clear(int i, int j) {
95          model[i][j] = 0;
96      }
97
98      /**
99       * checks if val is an acceptable value for the row i
100      */
```

```
101     private boolean checkRow(int val, int i) {
102         for (int rowN = 0; rowN < SIZE; rowN++) {
103             if (val == model[i][rowN]) {
104                 return false;
105             }
106         }
107         return true;
108     }
109 
110     /**
111      * checks if val is an acceptable value for the column j
112      */
113     private boolean checkCol(int val, int j) {
114         for (int columnN = 0; columnN < SIZE; columnN++) {
115             if (val == model[columnN][j]) {
116                 return false;
117             }
118         }
119         return true;
120     }
121 
122     /**
123      * checks if val is an acceptable value for the box around the cell
          at row i
124      * and column j
125      */
126     private boolean checkBox(int val, int i, int j) {
127         final int s = (int) Math.sqrt(SIZE);
128         /**
129          * square root of the size of the two arrays, resulting in 9
              which is
130          * the amount of boxes a typical Sudoku have
131          */
132         final int x = i / s * s, y = j / s * s;
133         //checks what box one is occupying in
134         for (int columnN = 0; columnN < s; ++columnN) {
135             for (int rowN = 0; rowN < s; ++rowN) {
136                 if (model[columnN + x][rowN + y] == val) {
137                     return false;
138                 }
139             }
140         }
141         return true;
142     }
143 }
```

6.2 Main

```
1  import javax.swing.JFrame;
2
3  /**
4   *
5   * @author r41
6   */
7  public class Main {
8
9      public static void main(String[] args) {
10         GUI window = new GUI();
11         window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12         window.setVisible(true);
13
14         Field field = new Field();
15         field.fromFile(args[0]);
16         try {
17             solve(field, 0);
18         } catch (SolvedException e) {
19             System.out.println(field);
20         }
21
22     }
23
24     public static void solve(Field f, int i) throws SolvedException {
25         if (i >= Field.SIZE * Field.SIZE) {
26             throw new SolvedException();
27         }
28
29         if (!f.isEmpty(i % Field.SIZE, i / Field.SIZE)) {
30             solve(f, i + 1);
31             return;
32         }
33
34         for (int x = 1; x <= Field.SIZE; ++x) {
35             if (f.tryValue(x, i % Field.SIZE, i / Field.SIZE)) {
36                 solve(f, i + 1);
37             }
38
39         }
40         f.clear(i % Field.SIZE, i / Field.SIZE);
41     }
42
43 }
```

6.3 Gui

```
1  import java.awt.BorderLayout;
2  import java.awt.Color;
3  import java.awt.Dimension;
4  import java.awt.Font;
5  import java.awt.GridLayout;
6  import java.awt.event.ActionEvent;
7  import java.awt.event.ActionListener;
8  import java.awt.event.MouseAdapter;
9  import java.awt.event.MouseEvent;
10 import java.io.FileWriter;
11 import java.io.IOException;
12 import java.util.logging.Level;
13 import java.util.logging.Logger;
14 import javax.swing.*;
15
16 /**
17  *
18  * @author r41
19  */
20 public class GUI extends JFrame {
21
22     public Field field;
23     public JButton checkbutton = new JButton("Check");
24     public JPanel area = new JPanel();
25     public boolean[][] fixed = new boolean[Field.SIZE][Field.SIZE];
26     //checker for true or false in the 9x9 arrays
27
28     public JMenu menu() {
29         JMenu file = new JMenu("File");
30         String names[] = {"open file", "save file"};
31
32         MouseAdapter actions[] = {
33             new MouseAdapter() { // Open file action
34                 @Override
35                 public void mousePressed(MouseEvent e) {
36                     JFileChooser fileChooser = new JFileChooser();
37                     if (fileChooser.showOpenDialog(null)
38                             == JFileChooser.APPROVE_OPTION) {
39                         load(fileChooser.getSelectedFile().getPath());
40                     }
41                 }
42             },
43             new MouseAdapter() { // Save file action
44                 @Override
45                 public void mousePressed(MouseEvent e) {
46
47                     for (int j = 0; j < Field.SIZE; j++) {
48                         for (int i = 0; i < Field.SIZE; i++) {
49                             if (field.isEmpty(j, i)) {
50                                 JOptionPane.showMessageDialog(null, "Sudoku
                                        needs to be finished, before saving is
```

```
                                an option.");
51                          return;
52                      }
53                  }
54              }
55
56              JFileChooser fileChooser = new JFileChooser();
57              fileChooser.setApproveButtonText("Save");
58              if (fileChooser.showOpenDialog(null)
59                      == JFileChooser.APPROVE_OPTION) {
60                  save(fileChooser.getSelectedFile().getPath());
61              }
62          }
63      }
64  };
65  for (int i = 0; i < names.length; ++i) {
66      JMenuItem j = new JMenuItem(names[i]);
67      j.addMouseListener(actions[i]);
68      file.add(j);
69  }
70  return file;
71
72  }
73
74  private void load(String path) {
75      field = new Field();
76      field.fromFile(path);
77      for (int j = 0; j < Field.SIZE; ++j) {
78          for (int i = 0; i < Field.SIZE; ++i) {
79              JTextField txt = (JTextField) area.getComponent(i + j *
                    Field.SIZE);
80              String str = Integer.toString(field.model[j][i]).trim();
81              txt.setText(str.equals("0") ? "X" : str);
82              txt.setHorizontalAlignment(JTextField.CENTER);
83              txt.setBackground(Color.WHITE);
84              fixed[j][i] = !field.isEmpty(j, i);
85          }
86      }
87
88  }
89
90  private void save(String path) {
91
92      final String space = " ";
93      try (FileWriter sv = new FileWriter(path, true)) {
94          sv.write(field.toString().replace("+", space).replace("-",
                space).replace("|", space).replace("0", "X"));
95      } catch (IOException ex) {
96          Logger.getLogger(GUI.class.getName()).log(Level.SEVERE, null,
                ex);
97      }
98  }
99
100 public GUI(String path) {
```

```
101         this();
102         this.load(path);
103     }
104 
105     public GUI() {
106         this.setLayout(new BorderLayout());
107         this.add(area, BorderLayout.CENTER);
108         this.add(checkbutton, BorderLayout.SOUTH);
109         checkbutton.addActionListener(new ActionListener() {
110             @Override
111             public void actionPerformed(ActionEvent e) {
112                 for (int j = 0; j < Field.SIZE; ++j) {
113                     for (int i = 0; i < Field.SIZE; ++i) {
114                         JTextField txt = (JTextField) area.getComponent(i +
                                j * Field.SIZE);
115                         if (fixed[j][i]) {
116                             txt.setText(Integer.toString(field.model[j][i])
                                    );
117                         } else {
118                             if (txt.getText().trim().toLowerCase().equals("
                                    x")) {
119                                 continue;
120                             }
121                             try {
122                                 int r = Integer.parseInt(txt.getText());
123                                 Color back = field.tryValue(r, j, i) ? Color
                                        .BLUE : Color.RED;
124                                 txt.setBackground(back);
125                             } catch (NumberFormatException ne) {
126                                 txt.setText("X");
127                                 JOptionPane.showMessageDialog(null, "Only
                                        accepts integers.");
128                                 return;
129                             }
130                         }
131 
132                     }
133                 }
134             }
135         });
136         final int dim = 50;
137         checkbutton.setPreferredSize(new Dimension(dim, dim));
138         JMenuBar bar = new JMenuBar();
139         bar.add(this.menu());
140         this.setJMenuBar(bar);
141         final int gap = 10;
142         area.setLayout(new GridLayout(Field.SIZE, Field.SIZE, gap, gap));
143         area.setBackground(Color.BLACK);
144         for (int i = 0; i < Field.SIZE * Field.SIZE; ++i) {
145             final int size = 25;
146             final int fsize = 25;
147             JTextField txt = new JTextField();
148             txt.setPreferredSize(new Dimension(size, size));
```

```
149             txt.setFont(new Font(txt.getFont().getName(), Font.PLAIN,
                    fsize));
150             txt.setBorder(javax.swing.BorderFactory.createEmptyBorder());
151             area.add(txt);
152         }
153         this.pack();
154     }
155 }
```

6.4 SolvedExceotion

```
1 /**
2  *
3  * @author r41
4  */
5 public class SolvedException extends Exception {
6
7 }
```

Litteratur

[1] Jan Baumbach

University of Southern Denmark

http://www.imada.sdu.dk/~jbaumbac/download/teaching/ws16-17/DM550/project/re-exam_introduction_to_programming_java_project.pdf