Design and Style Guide

Jouke van der Maas & Koen Keune

March 27, 2013

1 Overview

During this project, a sudoku solver is implemented. Only logical solving strategies are used; there are no brute-force methods. The underlying principle to the solve method is that once possible numbers have been calculated for each square, they can only be removed. Removal of possible numbers is achieved by a strategy. By trying strategies repeatedly, one of two states can always be achieved. From these states, a new number can be entered and the process can repeat.

1.1 End-states

The first end-state from which a new number can be entered is the *Single Possibility rule*. This rule states that if a given cell has only one possible number, that number should be entered. The second end-state is the *Only Square rule*. This rule states that if a given number can be entered in only one cell within a container (row, column or square), that number should be entered. All other strategies implemented for the project only remove possibilities from cells, leading to one of these two end-states.

2 Usage

2.1 SudokuApp

3 Implementation

3.1 Important types

The class **Sudoku** represents the puzzle. It holds **CellContainer**s that holds the individual **Cell**s. A **CellContainer** can either be a row, column or square within the sudoku. These can be represented using only one type, because their behaviour is exactly the same. The **Cell** class represents cells within the sudoku. This class can be in exactly one of two states; it can have a value or it can have a list of possible values. These two states were not separated in two subclasses of an abstract class for simplicities sake.

The **Sudoku** is used by the class **Solver**. This class takes a **Sudoku** and solves it. It uses implementations of the interface **Strategy**. This interface defines a strategy for removing possibilities from cells, leading to one of the two end-states.

3.2 Important methods

3.2.1 Sudoku

- `Sudoku(int[][])`
  The constructor takes a puzzle represented as a jagged array of ints (`int[][]`).
- `CellContainer[] getRows()`
  `CellContainer[] getColumns()`
  `CellContainer[] getSquares()`
  `CellContainer[] getAllContainers()`
  These methods return the rows, columns and/or squares ordered from left to right, top to bottom. The return values are arrays because the number of containers never changes.

3.2.2 CellContainer

- `List<Integer> getValues()`
  Returns a list of all values that have already been entered in this **CellContainer**.

- `Cell[] getCells()`
  Returns an array containing the cells in this container. This is an array because the number of cells never changes.

3.2.3 Cell

- `List<Integer> getPossibilities()`
  Returns a list containing all possibilities for this cell. Throws a **`SudokuException`** if the cell has a value.
- `boolean hasValue()`
  Returns `true` if this cell has a value, false otherwise.
- `boolean removePossibility(int)`
  `boolean removePossibilities(Collection<Integer>)`
  These methods remove one or more possible numbers from the list of possibilities. Throws a **`SudokuException`** if the cell has a value. Returns `true` if the possibilities were removed, `false` otherwise.
- `int getValue()`
  Returns the value if there is one. Throws a **`SudokuException`** if there is not.
- `void setValue(int)`
  Changes the state of the cell from not having a value to having a value. Throws a **`SudokuException`** if the cell already had a value.

3.2.4 Strategy

- `boolean removePossibilities(Sudoku)`
  Removes one or more possibilities from the cells in the provided sudoku. Returns `true` if possibilities were removed, `false` otherwise.

3.2.5 Solver

- `void solve()`
  Solves the sudoku provided in the constructor. The method loops over an array of **`Strategy`** implementations, calling each one on the sudoku. If a possibility was removed, it tries to fill in a number and starts again from the top of the list. The list is ordered so cheaper (in terms of time) strategies are at the top. This means that more complex strategies are only used if they are necessary. The method returns when the

sudoku is solved or when no more possibilities could be removed and no more numbers could be filled in.

4 Used Strategies

Following are the strategies that are used in this particular order based on complexity.

4.1 OneOfEach

Removes possibilities in a cell by looking at remaining possibilities in rows, columns and squares.

4.2 Locked

Searches for 2 or 3 possibilities in a column that are the same and don't occur in another place in the square if so remove those possibilities on every other place on the same column.

4.3 DoubleLocked

Searches for possibilities that occurs in one row (or column) in a square and another row (or column) in another square so that blocks possibilities in those rows (or columns) in the third square.

4.4 NakedGroup

4.5 HiddenTwin

4.6 ForcingTwin

Looks at the consequences of filling the possibilities of a cell with two possibilities, removes a possibility if that possibility will be removed in both cases.

5 Style Guide

The Oracle Code Conventions for Java are used as a guideline in naming of methods and types, commenting styles and code layout.