Sudoku

Technical Manual

Sudoku Team

Dunni Adenuga

Andrew Nyhus

Tim Woodford

Introduction

The Sudoku program aims to be an all-in-one solution to all (or at least most) 9x9 sudoku needs. 9x9 sudoku is a number game where the player begins with only part of a 9x9 grid filled in. The player must then fill in the remaining squares such that each row and column contain each number from 1 through 9 exactly once. In addition, the board is partitioned into 9 3x3 non-overlapping squares which a placed beginning in the upper left corner. Each square must contain the numbers 1 through 9 exactly once.

The program can be roughly partitioned into 4 components: the core data structures, the puzzle solvers, the board generator, and the GUI. The core data structures form the model for the GUI, while the GUI module itself contains the view and controller components. With the exception of the core data structures, the remaining components are loosely coupled, so it would be relatively easy to replace the GUI with a web interface, or change the puzzle solving or generation methods. A rough view of the interactions between these components is shown in Figure 1.

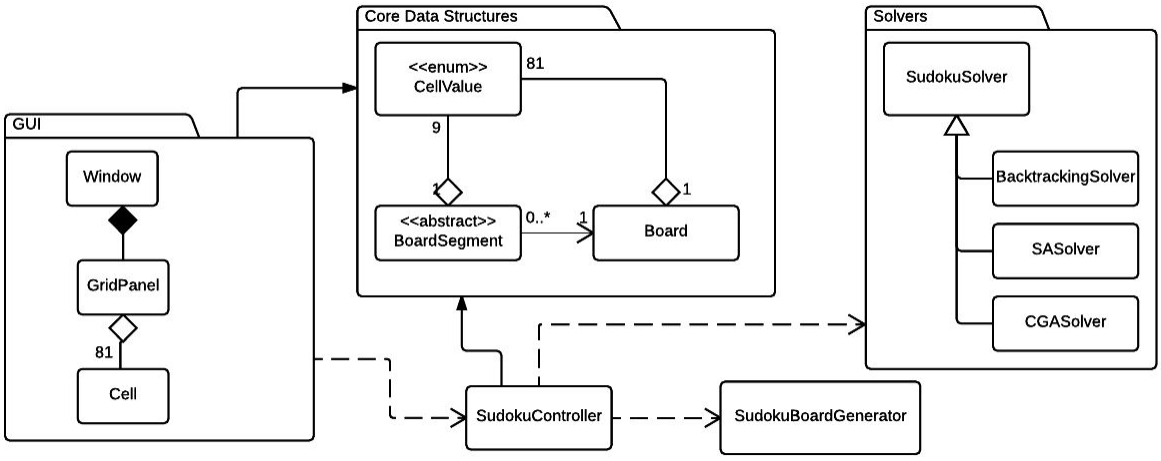

Figure 1: Coupling between modules

User Stories

The user stories we created for the program are the following:

- As a player, I want to be able to enter my own board, so that I can play the game or the computer can solve my puzzle for me.
- As a player, I want the computer to be able to generate the board for me with varying difficulty, so that I can play the game or the computer can solve the game.
- As a player, I want to be able to manually solve the puzzle.

- As a player, I want the computer to be able to solve the puzzle using three different algorithms.

The interactions between these user stories are shown in Figure 2.

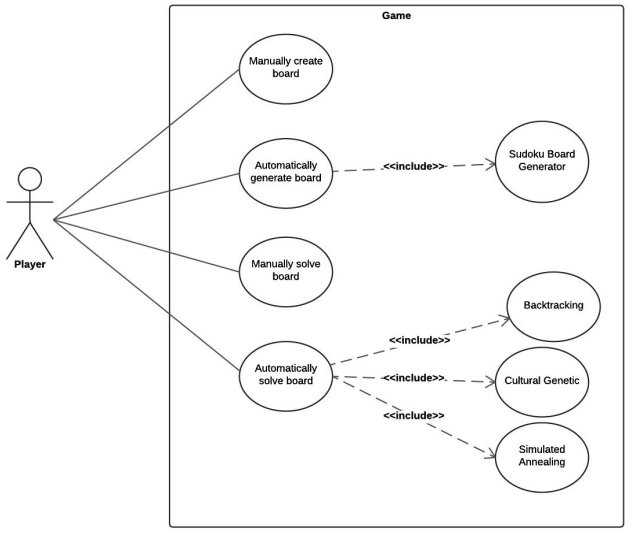

Figure 2: Use case diagram for Sudoku program

When the user selects the option to enter their own board, the controller clears the board and makes all of the squares on the board editable. After they are done setting up the board, they press a button which makes all of the filled squares non-editable and optionally begins running a timer.

Alternately, the computer can generate the board for the user. In this case, the controller calls the sudoku board generator, which returns a board that is then used for the view. The sudoku board generator is discussed further in the "Sudoku Board Generator" section below.

When the player manually solves the puzzle, they use a GUI that largely resembles the GUI used to enter custom boards, but with certain squares uneditable. This GUI uses a `Board` object as a model, which comes from either the board generator or from the user entering the board into the GUI.

Finally, the program can solve Sudoku boards using backtracking, simulated annealing, or cultural genetic algorithm, all of which can be used interchangeably as `SudokuSolver` objects. These algorithms will be discussed further in the "Sudoku Solvers" section below.

Design Overview

At the highest level of the program design, the CRC cards in Figure 3 show how the modules will interact. A more visual representation is shown in Figure 1 above.

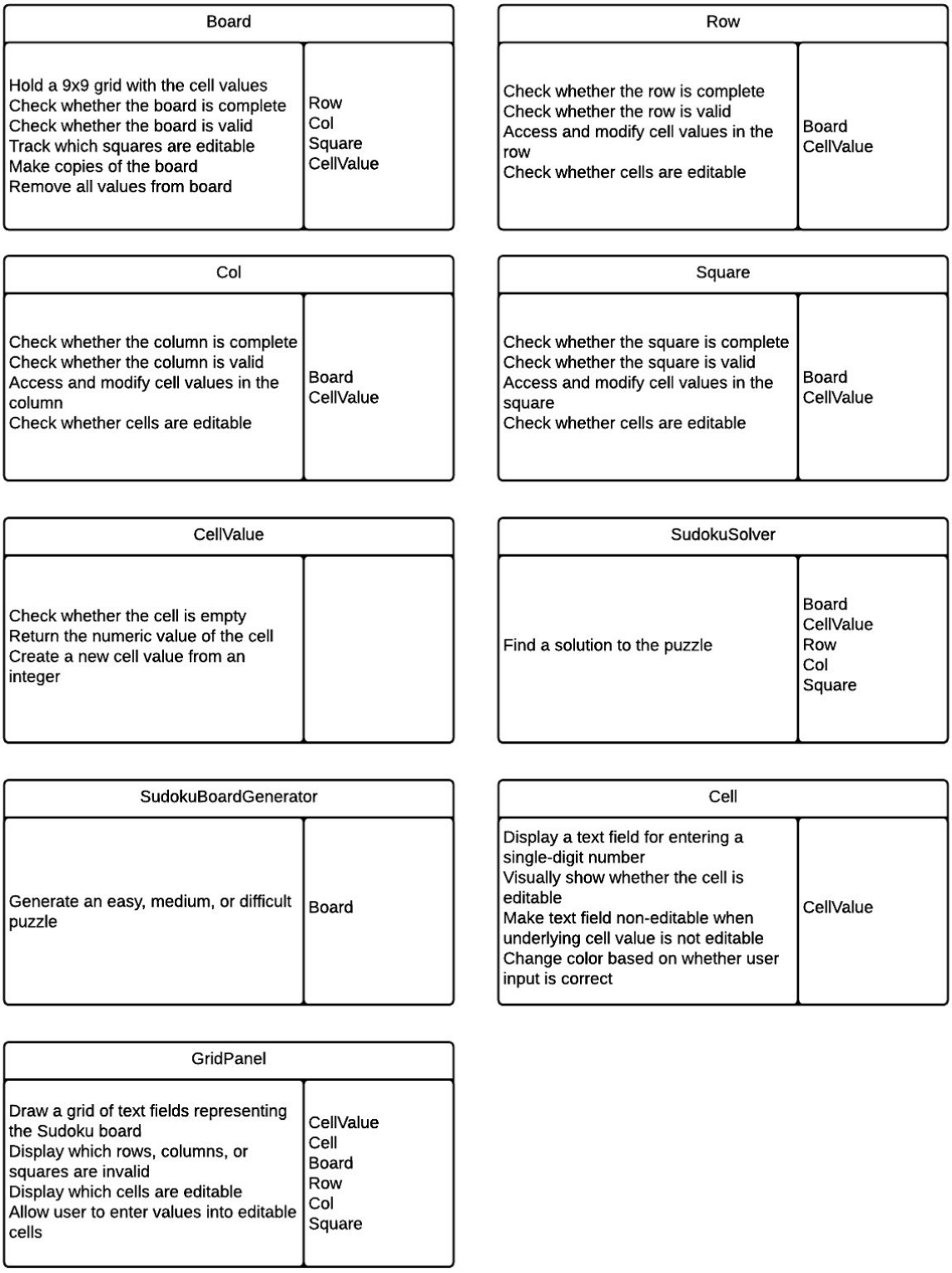

Figure 3: CRC Cards

The GUI operates with the `SudokuController` at the top. The `SudokuController` creates the Window class (custom JFrame), which includes the Menu Bar, the top panel (containing timer label and board progress label), and the GridPanel (custom JPanel). Inside the GridPanel is a 2-dimensional array of `Cell` objects (custom `JTextFields`). Any of the manipulation of the graphical sudoku grid is done through this GridPanel class. The GridPanel class is accessible through `Window.getGridPanel()`. The Cell class is provided with a pointer to the `SudokuController` object so that when the Cell is edited, gains/loses focus, etc. it can notify the `SudokuController`.

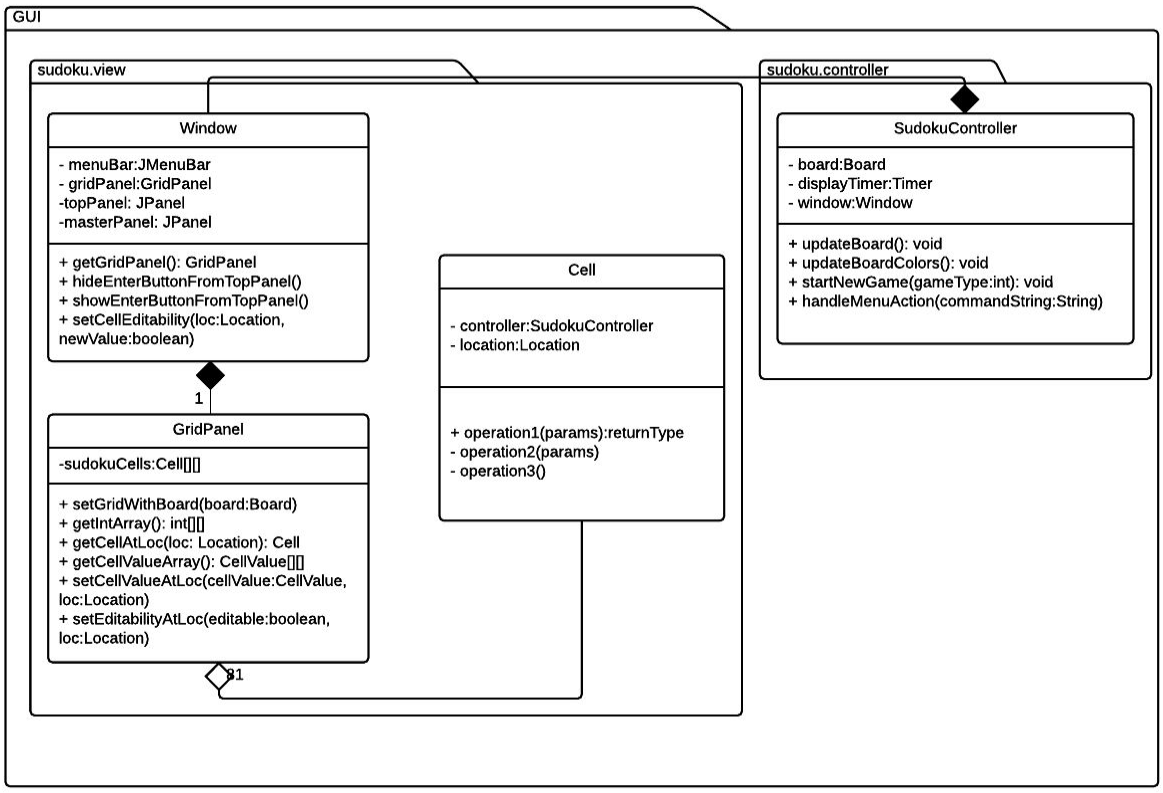

Figure 4: GUI UML Diagram

Core Data Structures

The core data structures contain a `Board` class and primitives for manipulating and evaluating sudoku boards in various stages of completion. The Row, Column, and Square class represent sets of 9 squares, in the rows, columns, and 3x3 squares in the puzzle. Specifically, we are interested in checking whether these segments are valid (there are no duplicate numbers in the segment), and whether they are complete (they contain all of the numbers 1-9 exactly once). These methods are used by the GUI to show errors while manually solving the board. To help in the implementation of solver algorithms, the segments also allow direct access to the numeric

values, editability values, and locations that make up the segment. With the exception of the Square class, the board segments need only implement a method to get the location of a square and the value of a square. The rest can be implemented by the abstract `BoardSegment` class. The segments reference the `Board` class directly, rather than a list of `CellValues`, so the segments are always up to date. The `Board` class then uses the board segments to check whether the board is complete or valid.

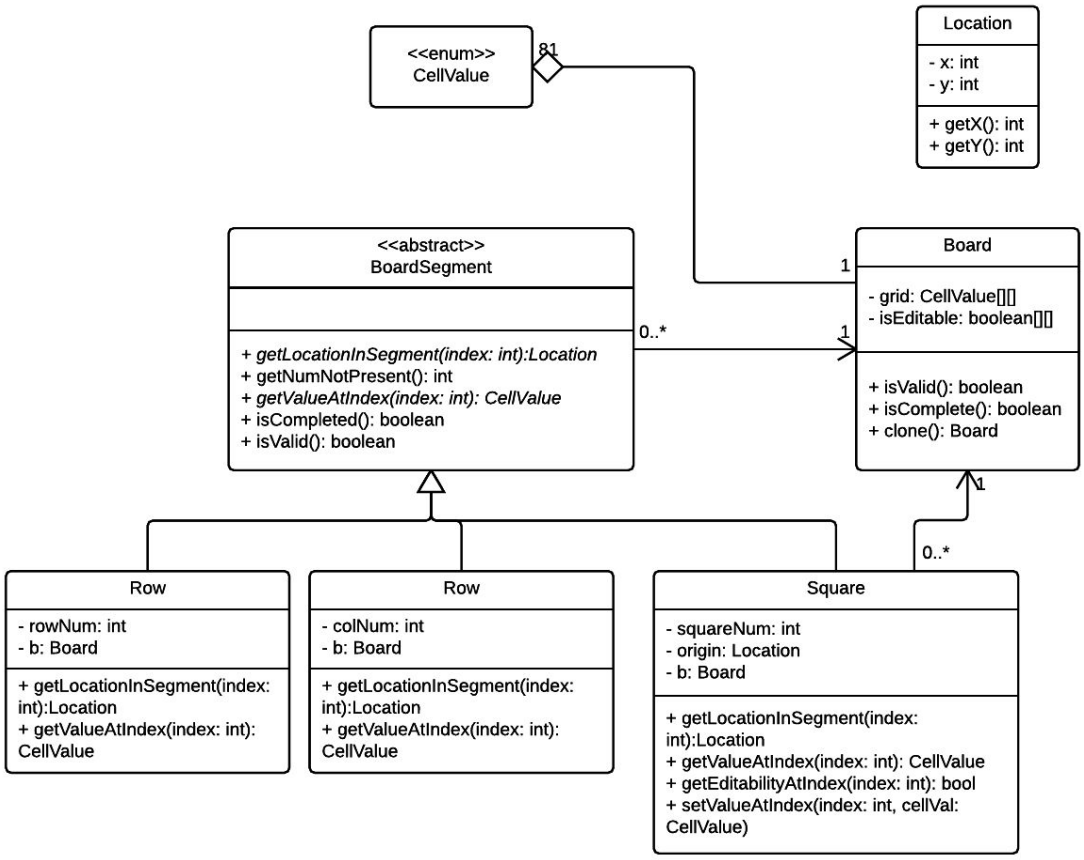

Figure 5: Core Data Structures UML Diagram

Solvers

The solvers package contains a number of algorithms that can be used to solve a Sudoku puzzle. All are represented as classes that implement SudokuSolver. The first is BacktrackAlgorithm, which is a self-contained class that can be used simply by calling the solveBoard() method. The Backtracking Algorithm is a brute force algorithm. It works through a mode of trial and error. It is also recursive. It finds the first empty cell, places a safe number(number appears once in row, column or 3x3 box). If the number placed into this first cell leads to a solution, that solution is reported, else, other safe numbers are tried and the process is reported.

The cultural genetic algorithm and simulated annealing solvers both involve additional classes. To understand how the interactions between the classes work, I will first present a high-level overview of the algorithms. Simulated annealing is a stochastic algorithm that works by first finding a random state for the editable squares. Then, during each iteration, two squares are swapped. The new board is compared to the previous board to determine the difference in how close the boards are to being correct. If the new board is closer, it is automatically used in the next iteration. If the old board is closer, the algorithm randomly decides whether to use the new board or the old board. The probability of choosing the new board is based on two factors: the difference in correctness between the two boards, and the current "temperature." Higher temperatures make it more likely that the new board will be chosen. After each iteration, the temperature is decreased by some factor. This is the "annealing" part of the algorithm.

The cultural genetic algorithm (CGA) begins similarly, with random state generation. Two of the best boards are then chosen, partially randomly. The two boards are "bred" together by using 3x3 squares from each board at random. Finally, there is a mutation step, which swaps two cells using the same process as simulated annealing.

Both the simulated annealing and CGA solvers have a similar structure: there is a general-purpose algorithm that can do any type of optimization, along with a specific implementation of the domain-specific state which allows the algorithm to solve Sudoku. In this was way, the algorithm implementation is decoupled from the exact optimization problem.

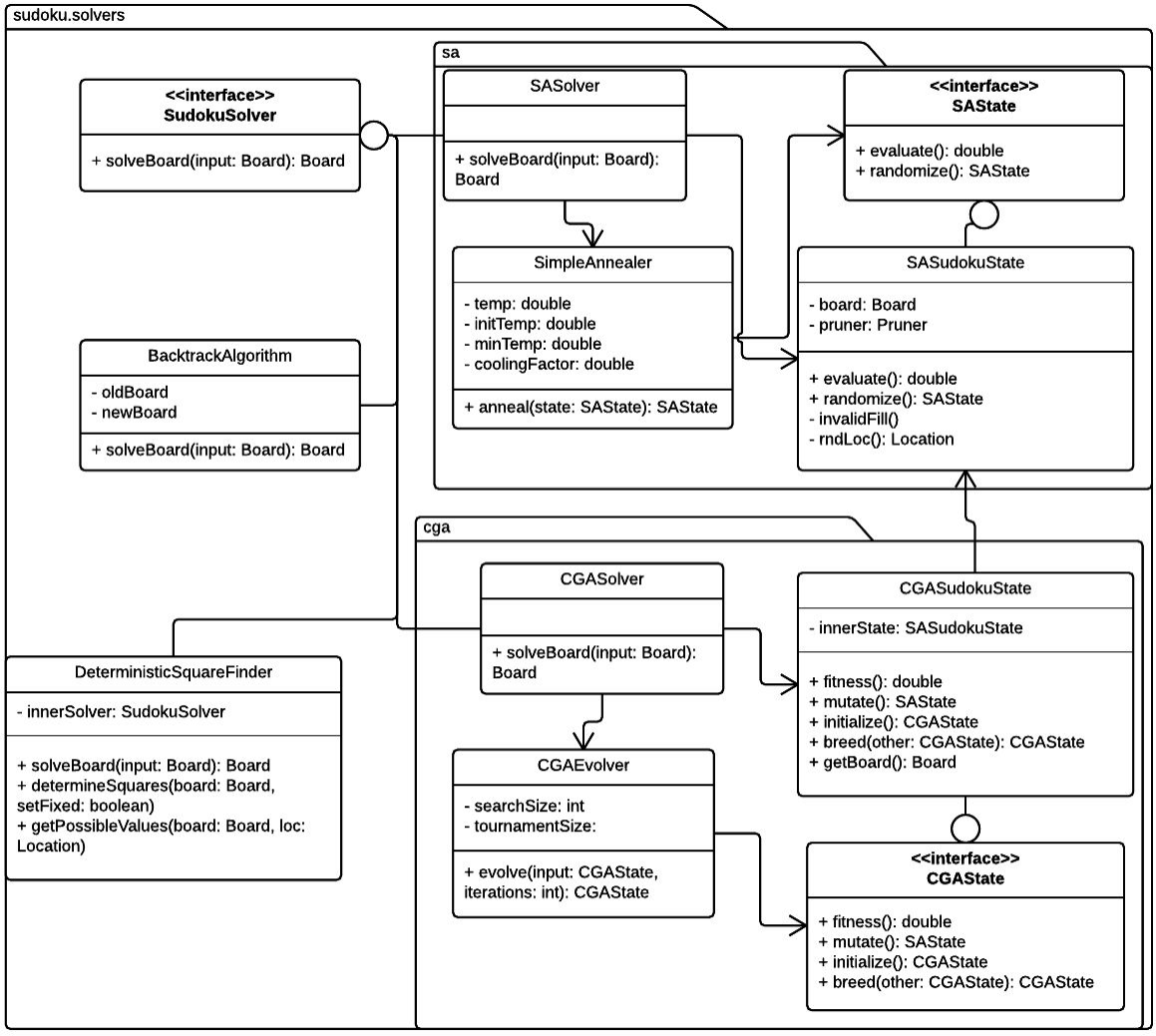

Figure 6: UML for the solvers module

Generating a Board

The `SudokuGeneratorBoard` class generates the sudoku puzzles. By calling the `generateBoard()` method, a sudoku puzzle of the Board class is returned.

The puzzle is generated by randomly choosing a solved board from a list of solved boards and then transposing the grid, swapping whole groups, swapping rows, swapping columns all in random order. Then, cells in the grid are erased from the grid randomly in a uniform manner.

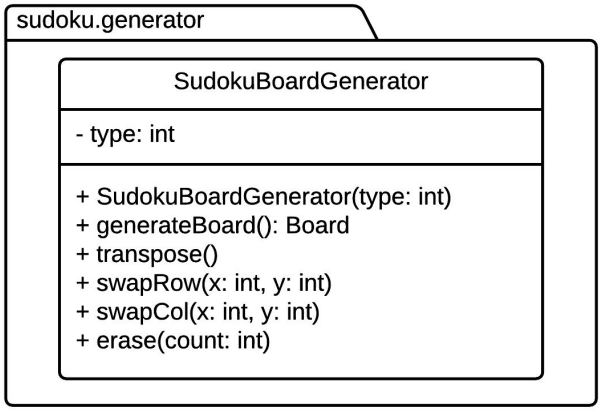

Figure 7: Board generator UML