Green Vs. Red

Project documentation
by Silvia Stoyanova

Content

1. Introduction

1.1 Green-Vs-Red game

‘Green vs. Red’ is a game played on a 2D grid that in theory can be infinite. Each cell on the grid can be either green (represented by 1) or red (represented by 0). The game always receive an initial state of the grid which is called “Generation Zero”. After that a set of 4 rules (1.1.1) is applied across the grid and those rules (1.1.1) form the next generation.

1.1.1 Rules

The following set of 4 rules forms the next generation. All the four rules apply at the same time for the whole grid in order for the next generation to be formed.
Rules:

- Each red cell that is surrounded by exactly 3 or exactly 6 green cells will also become green in the next generation
- A red cell will stay red in the next generation if it has either 0, 1, 2, 4, 5, 7 or 8 green neighbours
- Each green cell surrounded y 0, 1, 4, 5, 7 or 8 green neighbours will become red in the next generation
- A green cell will stay green in the next generation if it has either 2, 3 or 6 green neighbours

1.2 Program purpose

Given a particular cell and a generation number, the program counts in how many generations until the given one this cell was green.

1.2.1 Input/Output

The program accepts from user the following arguments:
x y - the size of the grid (**x** being the **width** and **y** being the **height**)
The next **y** lines should contain strings (long **x** characters) created by **0s** and **1s** which will represent the ‘Generation Zero’ state.
The last arguments should be coordinates (**x1** and **y1**) and the number **N**. **x1** and **y1** are the coordinates of the cell in the grid which we will inspect. The **N** argument is the number of the last generation which concerns us. The calculation includes Generation Zero and generation **N**.

The program should print the number of generations in which the (x1,y1) cell was green.

1.2.2 Constraints

- x <= y < 1000
- The grid is formed only by 0s and 1s
- Each new formed generation has the same size (x,y)
- Each cell can be surrounded by up to 8 cells - 4 on the sides and 4 on the corners. Exceptions are the corners and the sides of the grid.

1.2.3 Examples

Example1:

#3x3 grid, in the initial state, the second row is all 1s, how my times will the cell [1,0] (top center) become green in 10 turns?

```
3 3
000
111
000
1 0 10
#expected result: 5
```

The cell is green in the 1, 3, 5, 7, and 9 generations

Example2:

#4x4 grid, input:

```
4 4
1001
1111
0100
1010
2 2 15
#expected result: 14
```

The cell is green in all generations except for Generation Zero and generation 1.

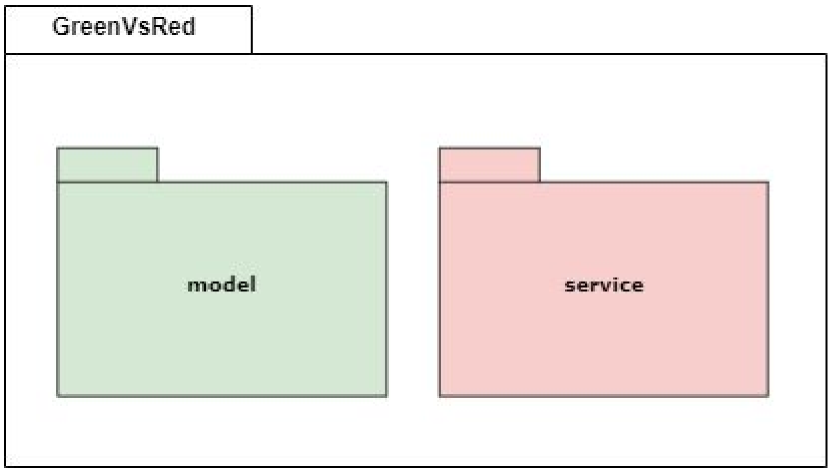
Example3:

```
3 4
111
010
100
011
2 1 4
#expected result: 2
```

The cell is green in 1 and 2 generations

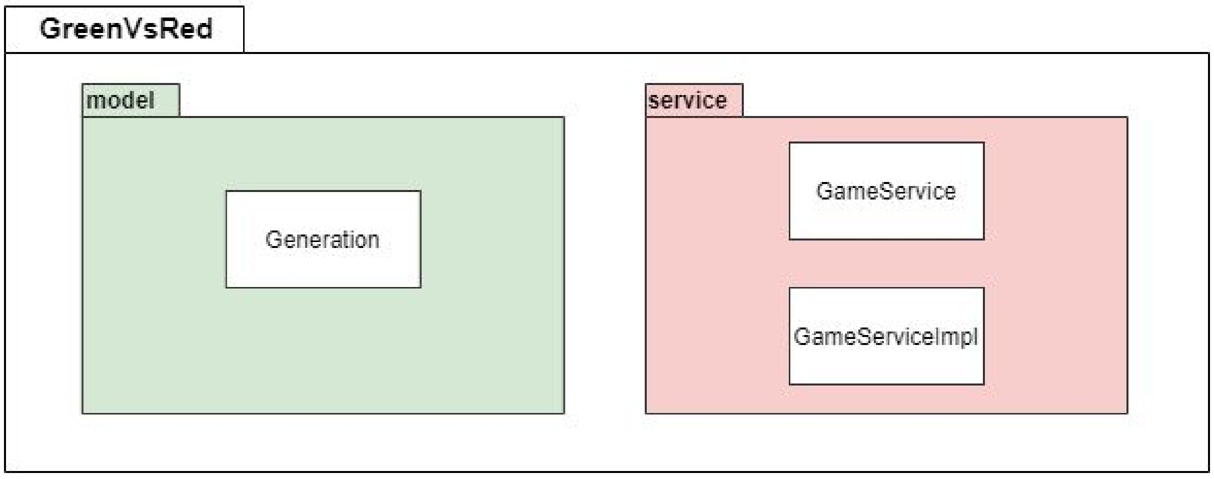
2. Decomposition

2.1 Packages



- **model** - contains the entities needed in order to complete the task
- **service** - contains the program logic

2.2 Interfaces and classes



2.2.1 Generation

Class Generation represents a single generation in the game. It stores the grid which is formed according to the rules.

Properties:

- **private int[][] grid -** the grid with 0s and 1s, which forms the generation
- **private int y** - the height of the grid (the number of rows)
- **private int x** - the width of the grid (the number of columns)

Constructors:

- **public Generation(int y, int x) -** x and y define the width and height of the grid

Methods:

- **public int getX() -** retrieve the width of the grid
- **public int getY() -** retrieve the height of the grid
- **public int getCell(int y, int x) -** retrieve a cell of the grid with given coordinates
- **public void setGreenCell(int y, int x)** - assign a cell of the grid with ‘1’ (make it green), given its coordinates

2.2.2 GameService

Interface GameService provides the methods needed in order to move to the next generations and follow the inspected cell.

Methods:

- **void goThroughGens()**
- **void printGreenStateCount()**
- **void updateGreenStateCount(Generation nexGeneration)**
- **void formNextGen(Generation nextGeneration)**
- **boolean becomesGreen(int y, int x)**
- **int countGreenNeighbours(int y, int x)**

2.2.3 GameServiceImpl

Class GameServiceImpl implements GameService. It defines the logic of the program.

Properties:

- **private int y1 -** the y coordinate of the inspected cell
- **private int x1** - the x coordinate of the inspected cell

- **private int numOfTurns** - the number of turns (generations excluding Generation Zero) we follow the inspected cell
- **private int greenStateCount** - the total count of turns (generations including Generation Zero) in which the inspected cell was green
- **private Generation currentGen** - the last formed generation

Constructors:

- **public GameServiceImpl(int x1, int y1, int turns, Generation generationZero) -** accepts **x1,y1** as the coordinates of the inspected cell, **turns** is the number of generations we follow the cell and **generationZero** is the initial generation which user defines in the input. The constructor sets **greenStateCount** to 1 if the inspected cell was green in Generation Zero, or 0 if it was red.

Methods:

- **void goThroughGens() -** goes through the generations until it reaches the last generation that concerns the user. After each iteration the method assigns to **currentGen** the newly formed one.
- **void printGreenStateCount() -** prints the result of program - the number of generations the inspected cell was green including Generation Zero and the last generation that concerns the user
- **void updateGreenStateCount(Generation nexGeneration) -** checks if the inspected cell is green in the newly formed generation and updates the **greenStateCount** if so. It is used in goThroughGens().
- **void formNextGen(Generation nextGeneration) -** forms the grid of the next generation. It is used in goThroughGens().
- **boolean becomesGreen(int y, int x) -** checks if the current cell would become green in the next generation. It is used in formNextGen().
- **int countGreenNeighbours(int y, int x) -** applies the rules of the game to count how many green neighbours has a cell in the current generation. It is used in becomesGreen().

All methods are inherited from interface GameService.

3. Additional information/Explanation

There isn't a setter for a red cell in the whole program, because the default value for int array is 0 (and red cells are also represented by 0)

While taking the input, the programs throws exceptions where Java doesn’t. This happens in case x >y or y <1000 (x < 1000 is already guaranteed with the first case) which violates the given constraints. It also throws exception if user tries to put a symbol different than 0 or 1 in the grid (otherwise the program would behave as if the cell was red).

If user has written a line longer than x, the program ignores the extra symbols. Probably throwing an exception here is also good.

The program keeps only the last formed generation because it is all we need to satisfy the task. In addition it could keep track of all passed generations linked together if we need to lookup to a previous one.

If a generation reached one of the following states:

- only 0s
- 2x2 subgrid of 1s and the rest is 0s

All of the next generations will be the same. In that case the iteration could be stopped earlier (now it is not).