Further Programming Assignment: Marupeke Version 1.3

Ian Wakeman

February 21, 2018

Changelog

1.1 Corrected name of parameters in randomPuzzle

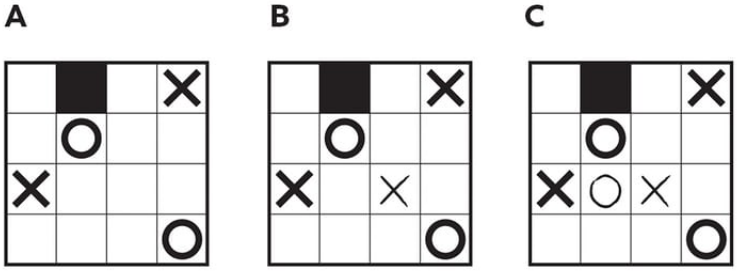
[**1.2**] Added suggestion for maximum grid size

1.3 Removed a confusing sentence, and provided further guidance in part 1.

Introduction

In this assignment, you will develop two versions of a Marupeke puzzle game. Marupeke is a Japanese puzzle, best described by Alex Bellos in the Guardian[1].

> Marupeke was invented in 2009 by Naoki Inaba, who is Japan's - and the world's - most prolific inventor of grid logic puzzles. "This puzzle is like a simple equation," he says. "In a simple equation two things lead to a third." In other words, when you add, subtract, multiply or divide two numbers, you get a third number. "I wanted to express this idea as simply as possible." It is also like playing noughts and crosses against yourself.
>
> **The rules**: Fill in each empty cell with either an O or an X, so that no more than two consecutive cells, either horizontally, vertically or diagonally, contain the same symbol.



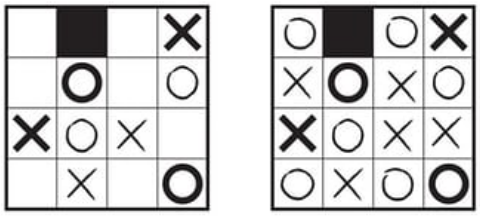
[1] `https://www.theguardian.com/technology/2017/oct/10/puzzle-masters-japan-sudoku-tokyo`

A: The starting grid.

B: The cell between the two Os must contain an X, otherwise there is a diagonal run of three Os, which is forbidden.

C: The cell between the Xs on the third row must contain an O, otherwise there is a horizontal run of three Xs, which again is forbidden.

D **E**



D: The cell on the bottom row beneath the Os in the second column must contain an X, otherwise there is a vertical run of three Os. Now you're on a roll. This new X threatens a diagonal run of three Xs, so there must be an O in the final cell of the second row. Continue using these strategies to complete the grid.

E: The completed grid.

In this computerised version, you will display a grid of tiles, some of which will be solid. You will then allow the user to mark a tile with an X or a O, and to check whether the current grid position is legal. In this assignment, you will first develop a textual version, and then a full graphical interface.

Important Dates

Part 1 handin 16:00, Thursday 1st March. Note that the maximum late period is 24 hours, and this submission is via study direct, not esubmission.

Part 2 handin 16:00, Thursday 12th April. Note that the maximum late period is 24 hours. Use esubmission.

Part 3 handin 16:00, Thursday 10th May. The maximum late period is 7 days. Use esubmission.

You must follow the instructions on submission naming and packaging. The usual late submission penalties will be applied to each part of the assignment.

Marks

The coursework is worth 50% of the marks for this course. The marks for the coursework will be divided as follows:

- Part 1 is 25% of the coursework total, with 80% for the work, and 20% peer assessment participation

- Part 2 is worth 25%
- Part 3 is worth 50%

The full marking schemes can be found with the description of each part.

Plagiarism, Collusion and Misconduct

As usual, the coursework you submit is supposed to have been produced by you and you alone. This means that on this course (the rules on other courses might possibly be different) you should not:

- work together with anyone else on this assignment,
- sit together in lab classes apparently working on on it separately, but in reality looking at how other people are doing it, and modifying your work accordingly. Note that this is not allowed, even if a tutor in the room does not notice!
- discuss details of the code for the assignment with other students.
- show code for the assignment to other students
- request help from external sources e.g. web-sites, forums, chat-sites, etc.
- do anything that means that the work you submit for assessment is not wholly your own work, but consists in part or whole of other people's work, presented as your own, for which you should in fairness get no credit.

If you need help ask your tutor. The University considers it misconduct to give or receive help other than from your tutors, or to copy work from uncredited sources, and if any suspicion arises that this has happened, formal action will be taken. Remember that in cases of collusion (students helping each other on assignments) the student giving help is regarded by the University as just as guilty as the person receiving help, and is liable to receive the same penalty.

Also bear in mind that suspicious similarities in student code are surprisingly easy to spot and sadly the procedures for dealing with it are stressful and unpleasant. Academic misconduct also upsets other students, who do complain to us about unfairness. So please don't collude or plagiarise!

Finally, if you do for any reason use code that was taken from elsewhere (e.g. from another student, a web-site, a text-book other than the course book, etc.) just be clear in the comments in the code, and in any write-up, exactly which bits of code were not produced by you. If you use code from elsewhere, and have modified it, be clear about what the modifications were. If you do this, you may or may not receive full marks for the assignment, but you will not be accused of any misconduct with subsequent penalties. Being honest is what all this is about!

1 Part 1: Storing a grid and recording marks and filled squares

The aim of Marupeke is to explore a grid of tiles, applying logic to discover how to mark the grid with noughts and crosses such that the rules of Marupeke are obeyed. We therefore start by designing a simple data structure to hold the solid squares and which squares have been marked with a nought or a cross.

The obvious data structure to use to represent the cells in the grid is a 2-dimensional array of chars, where we use 'O' and 'X' for noughts and crosses, and '#' for solid squares. If we declare:

```
char[][] grid = new char[10][10];
```

then we have our 10 by 10 grid.

Conventionally we think of the first co-ordinate as the row and the second as the column, so we think of grid [0][0] as the top-left square; grid [0][1] as the square just next to it; grid [1][0] as the first square in the second row; and so on, until grid [9][9] is the bottom-right. As well as the location of the mines, we need to determine whether a given square is editable - the starting grid will have filled out squares and noughts and crosses which cannot be removed. We will therefore use a 2-dimensional array of **boolean**s, declared as:

```
boolean[][] editable = new boolean[10][10];
```

1.1 Coding up and printing a Marupeke grid

Create a MarupekeGrid class with fields to store the grid and whether a square is editable, as above. Create a unit test to develop the following code using TDD. For each method, first create the tests that would fail against a correctly working method, write method, fail, correct the test, pass. Note that the `set*` methods below are intended to be used in two phases - first to initialise the grid with the starting state, so that after setting the value of the square, it should no longer be editable, and then in the user playing phase, where the user can change and remove marked squares as required. Write the following methods for this class:

- A constructor which takes as parameter the width of the grid. The constructor should create the initial arrays, up to a maximum width of 10. Write an existence and bounds checking test, although it won't be very interesting.
- A setSolid method with x and y parameters to set the given grid square solid. This call should also set the square to be uneditable. It should return **true** if the square was orginally editable, and **false** if the square was uneditable.
- setX and setO methods, which should have x and y parameters for the square in the grid to set, and a canEdit boolean parameter to change

the editable state of the square. It should return **true** if the square was orginally editable, and **false** if the square was uneditable.

- Create a static method to generate a random puzzle, with the following signature:

```
public static MarupekeGrid randomPuzzle(int size,
                                        int numFill,
                                        int numX,
                                        int numO);
```

  The squares to be filled, X or O should be randomly chosen to match the required numbers specified in numFill, numX and numO. If the sum of squares to be completed is greater than $size^2$, then you should return null. Note that these puzzles may be illegal, or uncompletable; we will determine how to generate a legal and finishable puzzle in the next two parts.

- Create a new toString method, which returns a string representing the grid, with the '#' character for a solid square, a 'O' for a nought and a 'X' for a cross. To identify empty squares, use the '_' (underscore) symbol. For instance, for a 4x4 Marupeke grid as in Figure A above, I would expect output like:

```
_#_X
_O__
X___
___O
```

Each method will be allocated a proportion of marks, with half of this proportion being awarded for a complete and correct test. You are allowed to write other methods to create elegant code, and to facilitate testing.

1.2 Submission of Part One

Your submission should be submitted through the Further Programming study direct assignment submission marked **Part One**. To simplify the marking process, you must submit your work as a Netbeans project, compressed as a standard zip file. Before submitting, create a clean directory, and ensure that your submission unzips to a Netbeans project that is usable. You should name this zip file with your candidate id eg as "92001893.zip". As far as practically possible, do not leave your name inside of any of your files - use your candidate id.

- If you use another compression tool, you will get zero marks.
- If you submit as any other IDE, you will get zero marks.
- If your submission is incorrectly named, you will get zero marks.

1.3 Marking of Part One

In the lab classes of week 5, we will do peer assessment of the submission for part 1. Each of you will be allocated two zip files, which you will mark against the model answer and the mark allocation scheme, completing a feedback sheet for each submission. These feedback sheets will then be returned to you via the office, and will form the basis for deciding whether to allocate the full marks or not. Participation in the peer assessment exercise will gain you 20% of the mark for this part, and thus 5% of your coursework mark.

2 Part 2: A textual game

In this part of the assignment, we will turn the Marupeke grid into a playable game from the command line. You will be supplied with the model answers from Part one.

You should create a separate project for part two, carrying over any classes that may be necessary.

2.1 Refactoring our classes

Holding information about a tile in two separate places is not elegant, and prone to confusion when we later try to maintain our code. Instead, we want to collect all information about a tile into one class. We are therefore going to refactor our code to use a single 2-dimensional array of MPTile classes. Our MPTile will have a **boolean** field to indicate whether it is editable, and an enumerated type to indicate whether it is blank, solid, an X or a O. You will then redesign the MarupekeGrid to utilise an array of MPTiles.

2.2 Marupeke by text

Having refactored the MarupekeGrid, we now want to add methods that allow people to play the game. Using test-driven development (write test methods first), create the following methods corresponding to user actions to the Marupeke class:

- **boolean** isLegal to return true if the grid as currently laid out is legal according to the rules above, and false if not.
- String [] illegalities should report the reasons why a grid is illegal if isLegal returns false.
- Modify randomPuzzle so that it will only return a legal puzzle. To ensure this is feasible, return **null** if the sum of the marked squares is greater then half of $size^2$.
- **void** markX and markY a tile at particular coordinates. This should succeed if the tile is editable.
- **boolean** unmark should unmark a square if it is editable, leaving it blank.
- **boolean** isPuzzleComplete method, returning true if all tiles are filled, and the grid is legal.

2.3 Getting commands from the user

In this version of the game, the user will type simple commands on the console input, and your program will execute actions in response to those commands. To simplify your work, I have provided a simple command line parser, which

returns the possible commands the user can enter. Your job is to integrate the parsing code into a control loop. Write a Marupeke class which creates a MarupekeGrid and then loops collecting commands from the user, calling the corresponding methods on the puzzle and displaying the updated puzzle to the user.

2.4 Part 2 Marking Scheme

Half the marks below will come from correct tests, and half from the code.

- 30% Refactored MPTile
- 30% Correct implementation of isLegal, illegalities methods and other supporting methods
- 20% Correct implementation of marking methods
- 20% Integration of command line parser to create workable game

2.5 Submission of Part Two

You should submit the assignment through esubmission, using the guidelines for zipping your project as for part one. Please ensure you use either standard compress or 7zip.

3 Part 3: The full graphical game

For this part, you are asked to write a class MarupekeGUI which gives a graphical user interface to your Marupeke class (or the one from the model answer to part two if you prefer).

3.1 Core requirements

Your JavaFX[2] GUI should consist of a window with an area which displays the Marupeke puzzle of the requisite size, and an area with other controls and displays, as follows.

- On starting your program, your GUI should display a random Marupeke grid. You should allow the user to choose a Marupeke puzzle of differing sizes and difficulty.
- The main area should give an up-to-date display of the grid, showing the uneditable tiles, and the current state of the editable tiles (eg through different colours). You are welcome to utilise the images on your tiles.
- The user should be able to mark a tile as X, O or unmark. Exactly how this should work is up to you. A common approach is to use a left click to cycle through the possible options. If the mark leaves the puzzle in an illegal state, this should be indicated to the user.
- At the conclusion of a game, either because the user has completed the tiles, or abandoned the game you should provide congratulations or commiserations, and the option to play again. You may want to investigate the use of dialogs for this purpose.
- In your Marupeke class, you should add a solve method using depth first search to determine whether a randomly generated grid is finishable or not. This should be used to ensure a finashable grid is set up initially. Optionally, you can use this method to help provide hints.

Take care to maintain a separation between the roles of the underlying Marupeke object, whose job is to store the grid data and provide methods to manipulate it, and the GUI code, whose job is merely to display that data and provide controls which link to the methods of Marupeke. You may need to add extra methods to the Marupeke class to make all this possible, and of course you are free to make whatever changes you like to that code.

3.2 Documentation

In addition to your code you should submit a README text file in your NetBeans project which explains how the code works, with the following (short) sections:

[2]You are welcome to use Swing if you wish to extend your learning.

Displaying the data: a section explaining how your GUI displays the data, how the data is fetched from the Marupeke object, and how the GUI is kept up to date.

Editing the data: briefly explain what happens when the user alters the data in the grid via the GUI

Optional Extras: give an account of any additional features you have implemented.

If you prefer you can put this writeup in another file (e.g. a Word document), but make sure that:

1. the file is in the same folder (directory) as your code so that it gets submitted along with everything else.
2. make sure the README file says what the name of this file is so that the person marking your work knows where to find the writeup.

3.3 Part 3 Mark Scheme

The percentage allocation of marks will be as follows:

20% for displaying the grid;

20% for the required functionality of marking a tile in the grid;

20% for completing the depth first search

10% for complete game functionality

30% for any additional functionality you add

3.4 Submission of Part Three

You should submit the assignment through esubmission, using the guidelines for zipping your project as for part one.