Java Implementation Skills

Oliver Wilkes

1. Overview

The goal of this assignment was to create a Java program that can generate a magic square to be solved by the user. The size of this magic square should be determined by the user, shuffled, and accept input to solve it. The program should also be able to check if the solution is correct and provide feedback to the user. My goal was to implement this functionality in a simple manner, so that the program can be extended later if needed.

2. Solution Design

My design consists of an entrypoint `MagicSquareGenerator` that contains the main method. This class is responsible for interfacing with the other class - `MagicSquare`, and validating initial user input. The generation algorithm is based off the Siamese method, which is a well-known algorithm for generating odd-order magic squares. When the user has made a swap, the program checks if the square is valid by sequentially checking each row and column for if the sum is equal to $\frac{n(n^2+1)}{2}$, where n is the size of the square. The program also checks the two diagonals for the same condition. If the user has made a valid swap, the program will print out the new square and ask if they want to continue. If they do not, the program will print out a message indicating that they have solved the square and exit.

The user's input is converted internally into a `MagicSquare.Direction`, which is an enum nested within the `MagicSquare` class to provide validation of user input, and simple selection for performing swaps. This class is `protected` so it can be accessed by the `MagicSquareGenerator` class, but not other packages. The grid of numbers is stored with a simple 2d array `int[][]` and the size of the square is stored in a `private int` attribute, so it cannot be modified outside of the class. If `size` was modified outside of `MagicSquare`, it would cause confusion as it would not be reflected in the resulting square's size.

3. Discussion

My solution is a simple implementation of a magic square generator and solver. Checking the entire validity of the square is simple but inefficient. This inefficiency is accpetable for the scope of this assignment, and for how little effect it has on the user experience. I had issues with the provided algorithm as it would not change column, so I had to research the algorithm for generating magic squares myself. One enhancement I delivered was the padding of numbers in the square only to the required length, so that the square is neat but not large. Another enhancement was the use of a `StringBuilder` to build the string representation of the square, which is more efficient than using string concatenation. An input of 1 for the size worked normally by saying the square has been solved in 0 moves, but I set the minimum size to 3 to avoid confusion.

4. Testing

I tested my solution initially by solving the puzzle myself, the ideal input. I then also tested the program with erroneous output and borderline cases, such as sizes of `-1`, `0`, `1`, `2`. During the testing of the main program. I ensured that swapping would wrap around borders as intended, and that invalid input is rejected with a message identifying the issue.